INTRODUCIÓN Á COMPILACIÓN

xoves, 1 de febreiro de 2024 09:13

O problema da tradución de código xorde dende a programación da primeira computadora en código máquina.

Neste tema veremos unha introdución aos **tradutores**, distinguindo os distintos tipos de tradutores e a súa evolución. Ademais, analizaremos a **compilación** como proceso paradigmático de tradución e as distintas fases da súa realización; o que servirá como resumo da materia.

Finalmente, introduciremos os **diagramas de Tombstone** como ferramenta de alto nivel para o deseño de estratexias de desenvolvemento de compiladores.

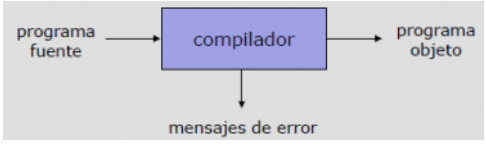
LINGUAXES DE PROGRAMACIÓN

- **Linguaxes máquina**: específicos de cada computadora, consisten en secuencias de 1s e 0s.
- **Linguaxes de ensambladores**: específicos de cada computadora, proporcionan un conxunto de nomes simbólicos para instrucións sinxelas da máquina (ADD, STO, ...) ou posicións da memoria (VALOR, MASA, ...).
- **Linguaxes de alto nivel**: independentes da máquina, incorporan estruturas de control, variables con tipo, aniñamento, procedementos, recursividade, tipos abstractos de datos, etc. (PASCAL, FORTRAN, COBOL, ...).
- **Linguaxes orientadas a problemas**: específicos de coleccións de problemas, reducen o tempo de programación, o seu mantemento e a súa depuración (SQL, GPSS, ...).

PROCESADORES DE LINGUAXES

compiladores

Un **compilador** é un programa que **traduce** outro programa, escrito nunha **linguaxe fonte**, a un programa equivalente escrito nunha **linguaxe obxecto**.

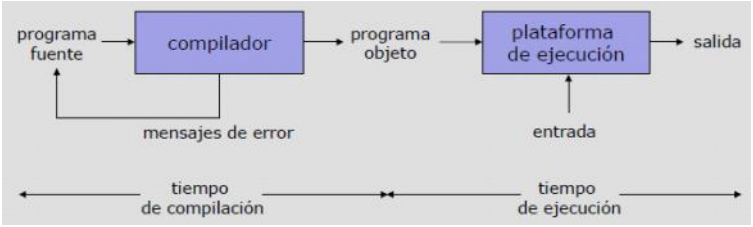


FORTRAN, COBOL, C, C++, PASCAL, ADA, ...

Ademais, o compilador informa da presencia de **erros** no uso da linguaxe fonte.

En xeral, o programa escrito na linguaxe obxecto é **executable** nunha computadora. Decimos entón que a linguaxe obxecto é unha **linguaxe máquina**.

Proceso completo de compilación e execución:



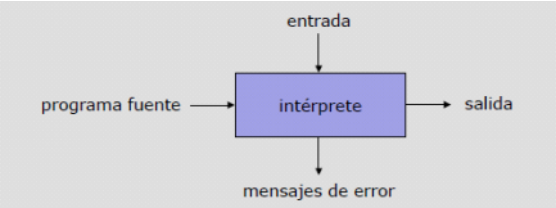
Pazo

PROCESADORES DE LINGUAXES

xoves, 1 de febreiro de 2024 09:36

intérpretes

Un **intérprete**, no lugar de producir un programa obxecto, aparenta **executar directamente** cada unha das instrucións do programa fonte, coas entradas proporcionadas polo usuario:



BASIC, LISP, PROLOG, SMALLTALK, APL, ...

PROCESADORES DE LINGUAXES

Compiladores vs. Intérpretes

Compiladores:

- → Compílase unha vez, execútase n veces.
- → A execución é máis rápida.
- → A xestión de erros abarca todo o programa.
- → Mellor adaptados a entornos de produción.

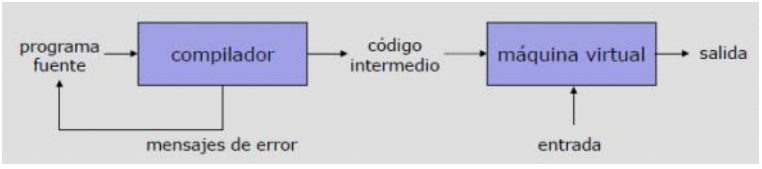
Intérpretes:

- → O programa tradúcese cada vez que se executa.
- → Necesita menos memoria.
- → Permítese a interacción co código en tempo de execución.
- → Mellor adaptados a entornos de desenvolvemento e experimentación.

PROCESADORES DE LINGUAXES

compilador-intérprete

Compilan o programa fonte dando lugar a un programa escrito nunha **linguaxe intermedia**. Despois, unha máquina virtual interpreta este programa:

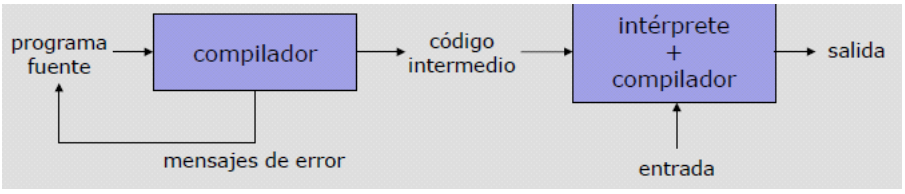


*Un compilador de Java xera un programa intermedio nun formato **bytecode**, que despois é interpretado por unha JVM. Tamén .NET ou PYTHON.*

PROCESADORES DE LINGUAXES

Compilador Just-in-Time

Compilan **en tempo de execución** fragmentos do código intermedio para dar lugar a un código obxecto, mellorando a velocidade de execución.



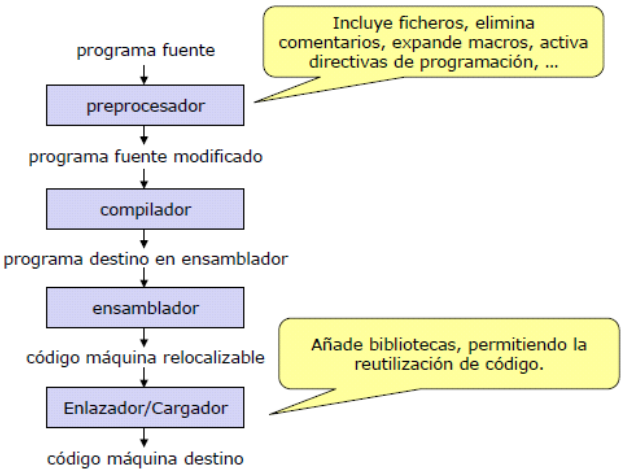
JAVA, .NET, PHP, PYTHON, ...

Mellora a **eficiencia** dos intérpretes e, en ocasións, incluso dos propios compiladores.

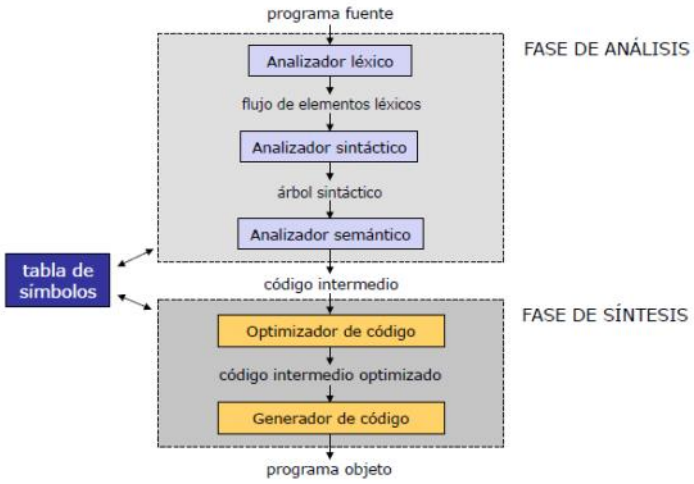
Pazo

O PROCESO DE TRADUCIÓN

xoves, 1 de febreiro de 2024 11:05



ESTRUTURA DUN COMPILADOR

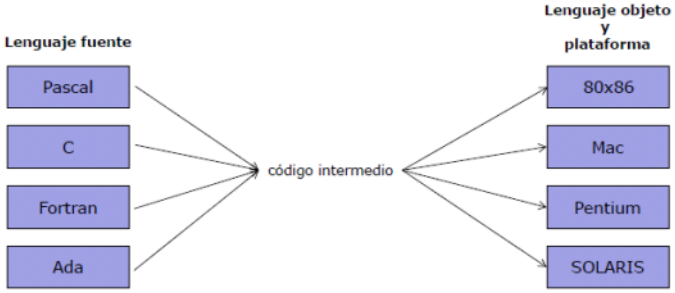


ESTRUTURA DUN COMPILADOR

estratexia

Supoñamos que se nos encomenda desenvolver un compilador para cada parella formada por unha linguaxe fonte e unha plataforma. Cántos artefactos software teríamos que desenvolver?

Se traballamos cun só tipo de código intermedio, só é necesario programas unha única fase de análise para cada linguaxe fonte e unha única fase de síntese para cada plataforma de execución.



ESTRUTURA DUN COMPILADOR

análise léxica

A análise léxica le o fluxo de caracteres que constitúen o programa fonte e agrúpaos en secuencias significativas, ou **compoñentes léxicos (*token*)**:

<nome, valor-atributo>

Exemplo:

```
posicion = inicial + velocidad *  60
```

→ <id, 1> <=> <id, 2> <+> <id, 3> <*> <num, 60>

táboa de símbolos

1	posicion	...
2	inicial	...
3	velocidad	...

Pazo

ESTRUTURA DUN COMPILADOR

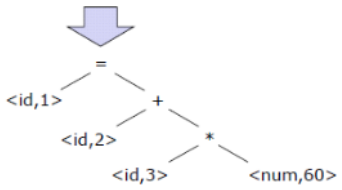
domingo, 4 de febreiro de 2024 17:00

análise sintáctica

A análise sintáctica crea unha representación intermedia en forma de árbore que describe a estrutura gramatical do fluxo de compoñentes léxicos. Nunha **árbore sintáctica**, cada nodo interior representa unha operación e os fillos do nodo representan os seus argumentos.

Exemplo:

<id, 1> <=> <id, 2> <+> <id, 3> <*> <num, 60>



ESTRUTURA DUN COMPILADOR

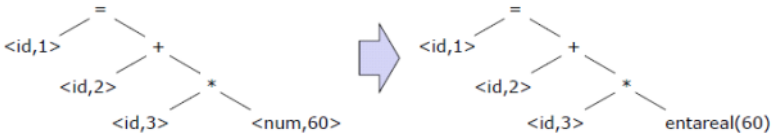
análise semántica

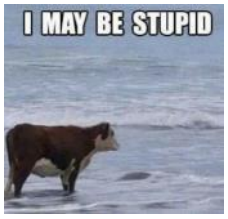
A análise semántica emprega a árbore sintáctica e a táboa de símbolos para comprobar a consistencia semántica do programa fonte.

Unha tarefa importante da análise semántica é a **verificación de tipos** admitidos pola linguaxe. Ademais, a linguaxe pode permitir certas **coercións** aos operandos.

Exemplo:

Supoñamos que todos os identificadores se declararon como reais.
Convertimos o número enteiro 60 a real. Coerción ao operador *.





ESTRUTURA DUN COMPILADOR

xeración de código intermedio

Podemos considerar o código intermedio como un programa para unha máquina abstracta. Debe ser fácil de producir e fácil de traducir ao programa obxecto.

Unha representación moi común do código intermedio é o **código de tres direccións**: secuencia de instrucións onde cada unha delas ten, como máximo, tres operandos.

Exemplo:

```
temp1 = entareal(60)
temp2 = id3 * temp1
temp3 = iid2 + temp2
id1 = temp3
```

Pazo

ESTRUTURA DUN COMPILADOR

domingo, 4 de febreiro de 2024 17:28

optimización de código

Esta fase consiste en **mellorar** o código intermedio: código máis rápido, máis curto, cun consumo menor de recursos, etc.

Exemplo:

Pódese deducir que a conversión de 60 de enteiro a punto flotante se pode facer en tempo de compilación.

```
temp1 = entareal(60)
temp2 = id3 * temp1
temp3 = iid2 + temp2      →      temp1 = id3 * 60.0
id1 = temp3                      id1 = id2 + temp1
```

ESTRUTURA DUN COMPILADOR

xeración de código

Tradúcese o código intermedio á linguaxe destino, polo xeral código máquina.

Selecciónanse rexistros de memoria para cada unha das variables declaradas no programa.

Exemplo:

Pódese deducir que a conversión de 60 de enteiro a punto flotante se pode facer en tempo de compilación.

```
temp1 = id3 * 60.0          →    LDF     R2, id3
id1 = id2 + temp1                MULF    R2, R2, #60.0
                                 LDF     R1, id2
                                 ADDF    R1, R1, R2
                                 STF     id1, R1
```

ESTRUTURA DUN COMPILADOR

táboa de símbolos

Todo compilador rexistra os **nomes** dos elementos do programa fonte (variables e procedementos), xunto cos seus **atributos**, que conteñen información de utilidade sobre cada nome.

- Exemplos de **atributos de variables**: a súa dirección de almacenamento, o seu tipo, a súa dimensión, o seu alcance, a súa precisión, se se declarou, se se inicializou, ...
- Exemplos de **atributos de procedementos**: o número e os tipo dos seus argumentos, o modo de paso (valor ou referencia), o tipo devolto, a recursividade, ...

CONSTRUCIÓN DE COMPILADORES

Na construción dun compilador hai que especificar as seguintes linguaxes:

- A **linguaxe fonte**.
- A **linguaxe obxecto** e a plataforma de execución.
- A **linguaxe de implementación**, na que está escrita o compilador.

Exemplo:

Dispoñemos dun compilador executable sobre PC, de Pascal a código máquina. A linguaxe fonte será o Pascal e a obxecto e a de implementación serán o código máquina do PC.

Pazo

CONSTRUCIÓN DE COMPILADORES

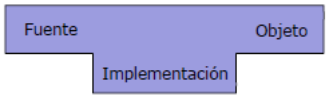
domingo, 4 de febreiro de 2024 18:17

diagramas de Tombstone

Os **diagramas de Tombstone** (ou diagramas en T) son unha ferramenta visual que facilita a comprensión do proceso de deseño de compiladores e intérpretes. Hai 4 tipos principais:

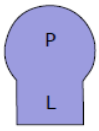
1. COMPILADORES:

Representa as tres linguaxes dun compilador:



2. PROGRAMAS:

Representa o programa P escrito en linguaxe L:

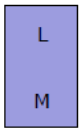


3. MÁQUINAS:

Representa a plataforma, sistema operativo ou computadora:

4. INTÉRPRETES:

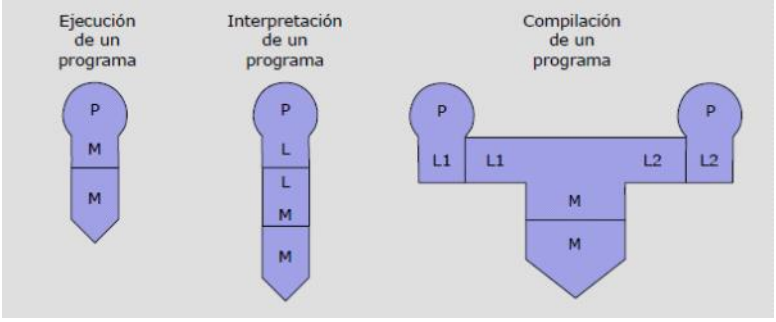
Representa o intérprete da linguaxe L escrito en M:



CONSTRUCIÓN DE COMPILADORES

unión de diagramas

Dous diagramas pódense unir se na unión as linguaxes son iguais.

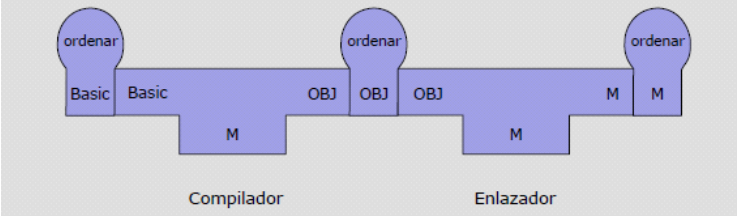


CONSTRUCIÓN DE COMPILADORES

compilador enlazador

Técnica que divide a **tradución en dúas fases**: a primeira traduce a linguaxe fonte a código intermedio e a segunda traduce o código intermedio a linguaxe máquina.

Emprégase para desenvolver múltiples compiladores para múltiples plataformas.



Pazo

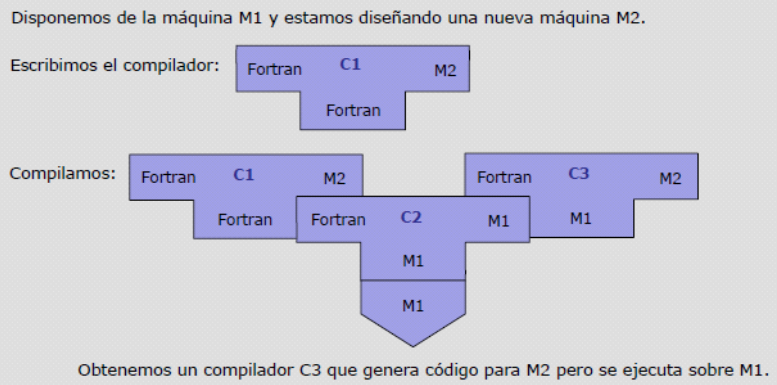
CONSTRUCIÓN DE COMPILADORES

domingo, 4 de febreiro de 2024 18:37

compilador cruzado

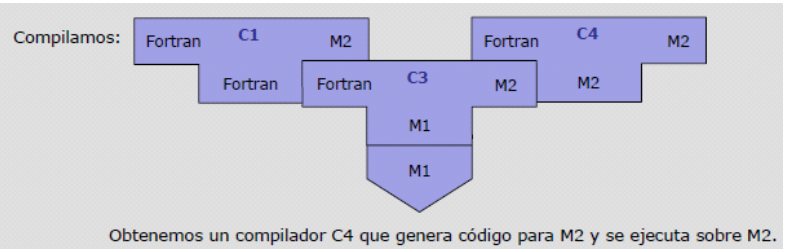
PRIMEIRA FASE

Técnica que permite desenvolver compiladores para máquinas diferentes á de desenvolvemento.



SEGUNDA FASE

Se compilamos unha segunda vez o compilador C1 co compilador creado C3, obtemos un compilador C4 que xera o código para a nova máquina M2 e se executa sobre M2.



CONSTRUCIÓN DE COMPILADORES

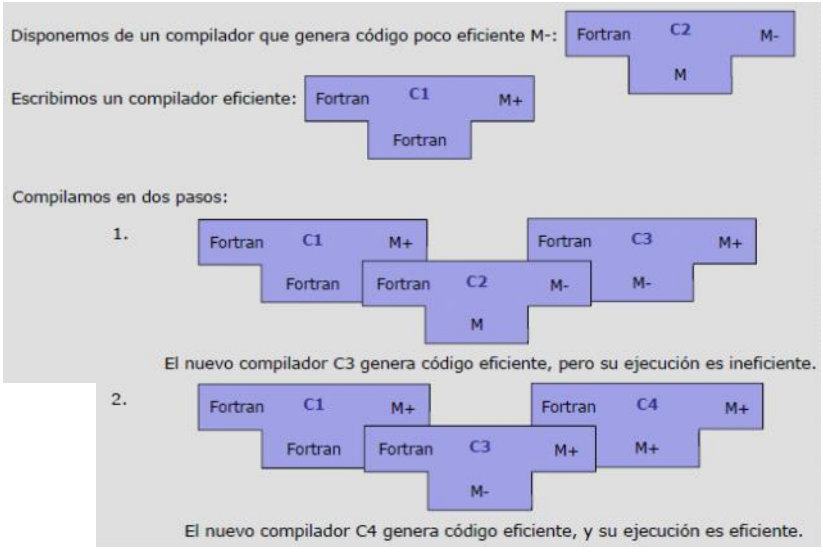
bootstrapping

Baséase en empregar múltiples **etapas** de compilación para mellorar a expresividade e o rendemento dun compilador.

Unha forma de bootstrapping consiste en construír o compilador dunha linguaxe a partir dunha versión reducida da mesma. Deste modo, obtéñense sucesivas versións que amplían e melloran unha linguaxe de programación determinada.

As maiores vantaxes do bootstrapping obtéñese cando un compilador se escribe na mesma linguaxe que compila. Un **autocompilador** é aquel que é capaz de compilar o seu propio código fonte.

Podemos aplicar bootstrapping á mellora da eficiencia dun compilador mediante sucesivas recompilacións.



CONSTRUCIÓN DE COMPILADORES

luns, 5 de febreiro de 2024 12:59

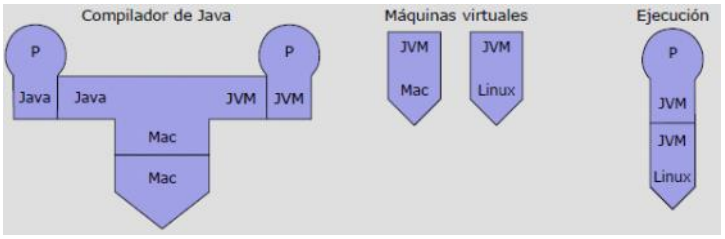
compilador-intérprete

Resulta da colaboración dun compilador e un intérprete co obxectivo de xerar un código intermedio fácilmente portable. A unión do intérprete e a plataforma de execución recibe o nome de máquina virtual.

Exemplo:

Exemplo:

A execución dun programa escrito en JAVA é un exemplo de proceso no que interveñen un compilador e un intérprete. O compilador dá lugar a un código intermedio sen erros, que se interpreta mediante unha JVM.



APLICACIÓNS

- **Edición de textos** con formato (*LaTeX*).
- **Recoñecemento de patróns**: de texto, fala ou visión por computadora.
- **Desenvolvemento de editores** de linguaxes estruturadas (*Xemacs*).
- **Cálculo simbólico** (*MAPLE*, *SCILAB*).
- **Deseño de circuítos integrados** (**mediante Verilog, VHDL**).
- **Tradución binaria** (*portar software entre plataformas*).
- **Simulación de arquitecturas de hardware**, para distintos conxuntos de datos, antes da súa fabricación.

Pazo

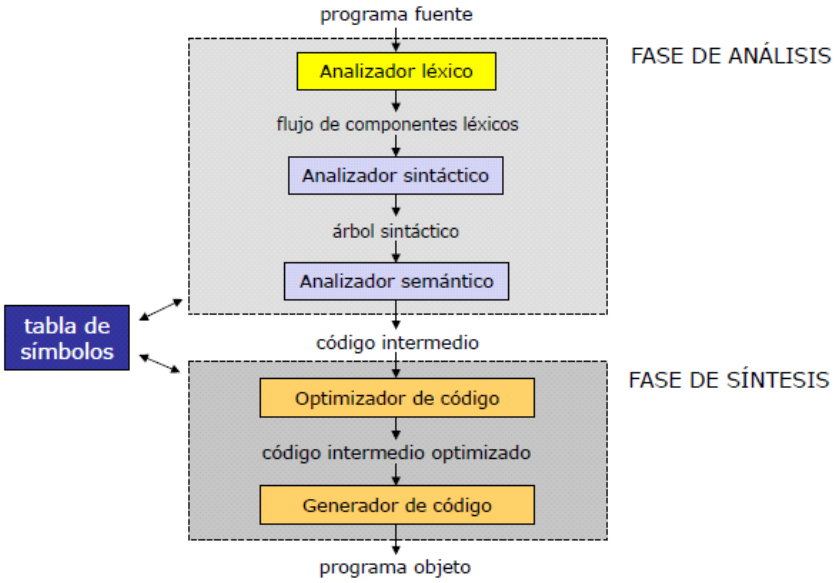
INTRODUCIÓN Á ANÁLISE LÉXICA

luns, 5 de febreiro de 2024 17:21

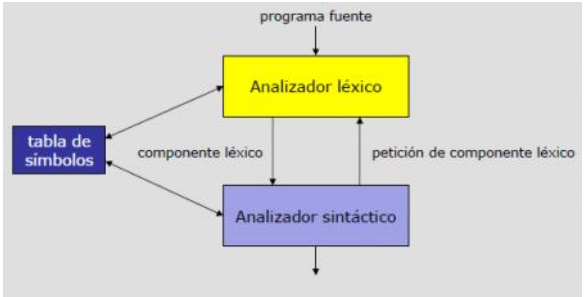
Neste tema estudiaremos a primeira fase do proceso de compilación: a análise léxica do programa fonte.

- Comezaremos **contextualizando** a análise léxica no proceso global de compilación.
- **Formalizaremos** esta análise mediante a **teoría de autómatas** e o uso das nocións de expresión regular, AFD e AFN.
- Exporemos o problema da xestión do código fonte mediante o estudo do **sistema de entrada**.
- Introduciremos a **táboa de símbolos**, que será de utilidade en todo o proceso de compilación.
- Introduciremos o problema do **tratamento de erros**.

ESTRUTURA



Os analizadores sintáctico e léxico funcionan segundo o patrón produtor-consumidor, seguindo o seguinte sistema:



Pazo

ESTRUTURA

xoves, 8 de febreiro de 2024 09:39

tarefas do analizador léxico

1) Recoñecer as **compoñentes léxicas** da linguaxe.
2) Eliminar aqueles caracteres do código fonte sen significado gramatical: os **espazos en branco**, os **caracteres de tabulación**, os **saltos de liña e de páxina**, ...
3) Eliminar os **comentarios** do programa fonte.
4) Recoñecer os **identificadores** de variable, tipo, constantes, métodos, etc. e gardalos na táboa de símbolos.
5) Avisar dos **erros léxicos** detectados e relacionar as mensaxes de erro co lugar no que aparecen no programa fonte (liña, columna, etc.).

ESTRUTURA

vantaxes de separar as análises léxica e sintáctica

- **Simplifícase a análise** en xeral. Illamos a análise sintáctica da chegada de caracteres inesperados, e así só ten que traballar coa estrutura da linguaxe. O deseño do compilador gaña en claridade.
- **Mellórase a eficiencia** do compilador. Podemos aplicar técnicas de mellora en cada fase.
- **Mellórase a portabilidade** do compilador. Se desexamos cambiar algunha característica do alfabeto de entrada, podemos cambiar o analizador léxico sen tocar o analizador sintáctico.

ESTRUTURA

deseño da análise léxica

1. Especificación dos **termos da análise léxica** mediante o uso de **expresións regulares**.
2. Deseño dun **autómata finito** que permita recoñecer as expresións regulares propostas.
3. Realización dun **autómata finto determinista mínimo** que permita realizar un recoñecemento eficiente.
4. Deseño e realización dun **sistema de entrada**.
5. Deseño e realización dunha **táboa de símbolos**.
6. Deseño e realización dunha estratexia de **manexo de erros**.

ESPECIFICACIÓN DO ANALIZADOR

termos da análise léxica

- **Compoñente léxica** (*token*): símbolo terminal da gramática que define a linguaxe fonte. Poden ser signos de puntuación, operadores, palabras reservadas, identificadores, ...
- **Patrón**: expresión regular que define o conxunto de cadeas correspondentes a unha compoñente léxica. *Así, '[0-9]+' identifica unha cadea de unha ou máis cifras, correspondente á compoñente léxica NÚMERO_ENTEIRO.*
- **Lexema**: cadea de caracteres presente no código fonte e que coincide co patrón dunha compoñente léxica. *Por exemplo, '456' coincide co patrón de NÚMERO_ENTEIRO.*
- **Atributos**: acompañan a cada compoñente léxica atopada e permiten a súa identificación e análise posterior. Na práctica, un único atributo apunta a unha entrada da táboa de símbolos.

Exemplos:

Componente léxico	Patrón	Lexemas
IDENTIFICADOR	[A-Za-z][A-Za-z0-9]*	imax, velocidad, rpm
NÚMERO_ENTERO	[0-9]+	245, 0, 9384100
MAYOR_O_IGUAL	>=	>=

A sentencia FORTRAN `E = C ** 2` *tradúcese como:*

1. *<IDENTIFICADOR, apuntador na táboa de símbolos á entrada E>*
2. *<OP_ASIGNACION>*
3. *<IDENTIFICADOR, apuntador na táboa de símbolos á entrada de C>*
4. *<OP_EXPOÑENTE>*
5. *<NÚMERO_ENTEIRO, valor enteiro 2>*

Pazo

ESPECIFICACIÓN DO ANALIZADOR

xoves, 8 de febreiro de 2024 12:34

expresións regulares

Unha expresión regular *r* permite representar patróns de caracteres. O conxunto de cadeas representado por *r* recibe o nome de linguaxe xerado por *r*, e escríbese L(r).

Para un alfabeto Σ diremos que:

1. O símbolo ∅ (conxunto baleiro) é unha expresión regular e L(∅) = {}.
2. O símbolo ε (palabra baleira) é unha expresión regular e L(ε) = {ε}.
3. Calquera símbolo a ∈ Σ é unha expresión regular e L(a) = {a}.

A partir destas expresións regulares básicas poden construírse expresións regulares máis complexas aplicando as seguintes operacións:

- **Concatenación**: Se *r* e *s* son expresións regulares, entón *rs* tamén o é; e $L(rs) = L(r) \cap L(s)$.
- **Unión**: Se *r* e *s* son expresións regulares, entón *r|s* tamén o é; e $L(r|s) = L(r) \cup L(s)$.
- **Cierre ou clausura**: Se *r* é unha expresión regular, entón r^* tamén o é; e $L(r^*) = L(r)^*$.

ESPECIFICACIÓN DO ANALIZADOR

definicións regulares

Por conveniencia, darémoslle nome ás ER e empregarémolos coma se fosen símbolos: $d \rightarrow r$

ESPECIFICACIÓN DO ANALIZADOR

simplificando a notación

- **Cero ou un caso**: O operador unitario posfixo *?* Significa "cero ou un caso de".
- **Un ou máis casos**: O operado unitario posfixo *+* significa "un ou máis casos de".
- **Clases de caracteres**: A notación *[a-z]* designa a ER a|b|...|z.

AUTÓMATAS FINITOS

AFN

Utilizaremos autómatas finitos no **recoñecemento de ER**. Un AFN é unha quíntupla (Q, Σ, δ, q_0, F) onde:

- Q é un conxunto finito de estados.
- Σ é o alfabeto de entrada.
- δ é a función de transición definida como $\delta: Q \times \Sigma \rightarrow P(Q)$
- q_0 ∈ Q é o estado inicial.
- F ⊆ Q é o conxunto de estados finais.

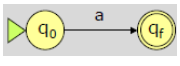
AUTÓMATAS FINITOS

deseño dun AFN

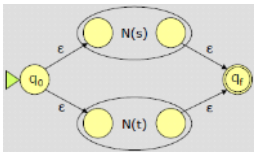
Cómo construír un AFN dun modo **sistemático** (é dicir, programable) a partir dunha ER?
Empregaremos a denominada **Construción de Thompson**, que utiliza tres regras básicas:

1) Para o símbolo ε, constrúese o AFN seguinte:

2) Para o símbolo a ∈ Σ, constrúese o AFN seguinte:

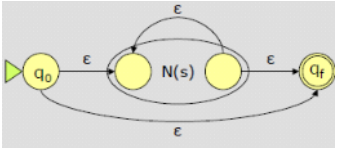


3) Supoñamos que N(s) e N(t) son AFN para as ER *s* e *t*:
   a) Para a ER s|t, constrúese o AFN N(s|t) seguinte:



   b) Para a ER st, constrúese o AFN N(st) seguinte:

   c) Para a ER s*, constrúese o AFN N(s*)seguinte:



Pazo

AUTÓMATAS FINITOS

xoves, 8 de febreiro de 2024 17:21

deseño dun AFN

Pódese demostrar que a aplicación sistemática das regras anteriores dá lugar a un AFN N(r) que recoñece a ER inicial r. Este AFN ten as seguintes propiedades:

- N(r) ten un estado inicial e un estado final. Esta propiedade satisfana tamén todos os AFN que o compoñen.
- N(r) ten, como moito, o dobre de estados ca de símbolos e operadores en r: cada vez que se aplica unha regra créanse como moito dous estados novos.
- Cada estado de N(r) ten unha transición saínte cun símbolo a ∈ Σ ou, como moito, dúas transicións saíntes con símbolos ε.

AUTÓMATAS FINITOS

AFD

Empregaremos AFD para un recoñecemento eficiente das ER. Un AFD é unha quíntupla (Q, Σ, δ, q_0, F) onde:

- Q é un conxunto finito de estados.
- Σ é o alfabeto de entrada.
- δ é a función de transición definida como $\delta: Q \times \Sigma \rightarrow Q$
- q_0 ∈ Q é o estado inicial.
- F ⊆ Q é o conxunto de estados finais.

AUTÓMATAS FINITOS

AFD equivalente a un AFN

Empregaremos o método de **construción de subconxuntos** para obter o AFD equivalente a un AFN dado. Este AFD será máis sinxelo de programar.

→ Na táboa de transicións dun AFN, a cada entrada correspóndelle un conxunto de estados. Na de un AFD, a cada entrada correspóndelle só 1.

→ A **idea** deste método é que a cada estado do AFD lle corresponde un conxunto de estados do AFN equivalente.

→ O AFD emprega cada estado para localizar todos os posibles estados nos que pode estar o AFN equivalente despois de ler cada símbolo de entrada.

Sexa o AFN N = (Q^N, Σ, δ^N, ${q_0}^N$, F^N), desexamos obter un AFD equivalente D=(Q^D, Σ, δ^D, ${q_0}^D$, F^D). Para isto, empregaremos as **tres operacións** seguintes:

- **cierre-ε(q)**: é o conxunto de estados do AFN n que se poden acadar dende o estado q ∈ $\mathrm{Q^D}$ con transicións ε só.
- **cierre-ε(Q)**: é o conxunto de estados do AFN N que se poden acadar dende os estados que pertencen ao conxunto Q ⊂ $\mathrm{Q^N}$ con transicións ε só.
- **move(Q, σ)**: é o conxunto de estados do AFN N aos que lles chega unha transición dende os estados que pertencen ao conxunto Q ⊂ $\mathrm{Q^N}$ co símbolo $\sigma \in \Sigma$.

E seguiremos os **pasos**:

1. Antes de detectar o primeiro símbolo de entrada, N pódese atopar en calquera dos estados do conxunto cierre-ε(${q_0}^N$). Polo tanto, chamarémoslle ${q_0}^D$ ao conxunto de eses estados Q ∈ $\mathrm{Q^N}$ tales que q ∈ cierre-ε(${q_0}^N$).
2. Dende calquera estado ${q_i}^D$ hai unha transición a un estado ${q_j}^D$ co símbolo $\sigma \in \Sigma$. Calculamos este estado en dous pasos:
   1) Primeiro, calculamos no AFN N o conxunto move(${q_i}^D$, σ) (recordemos que ${q_i}^D \subset \mathrm{Q^N}$).
   2) Despois, calculamos o seu cierre-ε.

   En definitiva, ${q_j}^D$=cierre-ε (move(${q_i}^D$, σ)).
3. No AFD D, un estado será final se contén algún estado final de AFN N.

Pazo

AUTÓMATAS FINITOS

xoves, 8 de febreiro de 2024 20:23

AFD mínimo equivalente

Un analizador léxico será máis **eficiente** cando menor sexa o número de estados do AFD correspondente. Para calquera AFD existe un AFD mínimo equivalente.

A idea é identificar pares de **estados equivalentes**: aqueles nos que o AFD se comporte igual para calquera cadea que detecte a partir deles.

AUTÓMATAS FINITOS

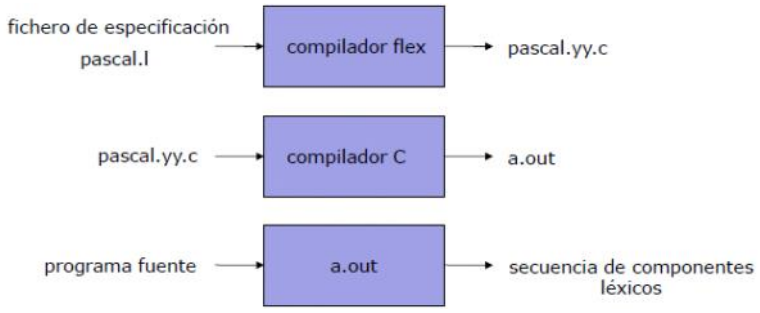
construción do analizador

O analizador léxico construirase a partir da agregación de AFD organizados segundo unha orde convinte.

A FERRAMENTA FLEX

Existen ferramentas que **xeran analizadores léxicos de forma automática** a partir dunha especificación baseada no uso de ER: por exemplo, **flex**.

O esquema de operación de flex é o seguinte:



A FERRAMENTA FLEX

o ficheiro de especificación

O **ficheiro de especificación** permite describir as compoñentes léxicas da linguaxe fonte e as accións a realizar. A compilación deste ficheiro dá lugar a unha función `yylex()` que invoca o analizador sintáctico para ler compoñentes léxicas.

Este ficheiro é da seguinte forma:

```
sección de definiciones

%%

sección de reglas de traducción

%%

sección de rutinas auxiliares
```

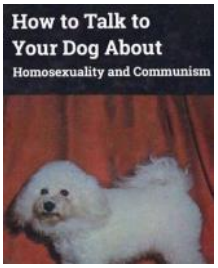
```
delim           [ \t\n]
espacio         {delim}+
letra           [A-Za-z]
digito          [0-9]
id              {letra}({letra}|{digito})*

%%

{eb}
if              {return(IF);}
then            {return(THEN);}
else            {return(ELSE);}
{id}            {yylval = inserta_tabla(); return(ID);}

%%

inserta_tabla() {/* instala el componente léxico en la tabla de
   símbolos, y devuelve un apuntador a él */}
```

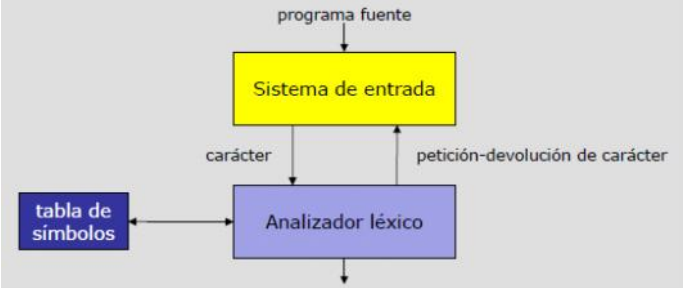


Pazo

O SISTEMA DE ENTRADA

xoves, 8 de febreiro de 2024 20:57

O **sistema de entrada** é un conxunto de **rutinas** que interactúan co sistema operativo para a lectura de datos do programa fonte. O sistema de entrada e o analizador léxico funcionan segundo o patrón produtor-consumidor, seguindo o seguinte esquema:



A separación do sistema de entrada supón unha mellora en:

- **Eficiencia**: supoñamos un analizador léxico en C que ten que acceder a un caracter dun ficheiro. Este caracter pasa 1) do disco á memoria xestionada polo sistema operativo, 2) a unha estrutura `FILE`, 3) a unha variable `string` do analizador. Urxe solucións.
- **Portabilidade**: o sistema de entrada é o único compoñente do compilador que se comunica co sistema operativo. Para cambiar de plataforma só temos que cambiar o sistema de entrada.

O SISTEMA DE ENTRADA
a memoria intermedia

O analizador léxico debe detectar a compoñente léxica co lexema máis largo posible. Pensemos nos exemplos ">"e ">=", ou "=" e "==". Necesitamos **ler caracteres dun modo anticipado**.

Para resolver este problema, débese incorporar unha **memoria intermedia** de modo que:

- Se poida almacenar un bloque de caracteres de disco e apuntar o fragmento xa analizado.
- En caso de devolución de caracteres ao fluxo de entrada, se poida mover un apuntador tantas posicións como caracteres a devolver.

O SISTEMA DE ENTRADA
métodos de xestión da entrada

Calquera sistema de entrada debe satisfacer:

→ Ser o máis rápido posible.
→ Permitir un acceso eficiente a disco.
→ Facer un uso eficiente da memoria.
→ Soportar lexemas de lonxitude considerable.
→ Ter dispoñibles o lexema actual e o anterior.

Estudaremos dous **métodos** de xestión do sistema de entrada:

- Método do **par de memorias intermedias** (*buffer*).
- Método do **centinela**.

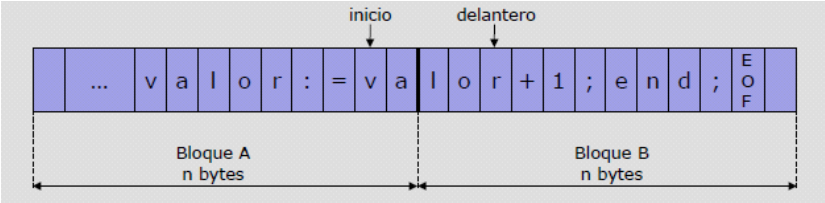
Pazo

O SISTEMA DE ENTRADA

xoves, 8 de febreiro de 2024 21:27

método do par de memorias intermedias

Este método divide a memoria intermedia en dúas metades de n bytes cada unha. N debe ser múltiplo da lonxitude da unidade de asignación.



Ao principio, os apuntadores **inicio** e **dianteiro** apuntan ao primeiro caracter dun lexema. A medida que o analizador pide caracteres, dianteiro móvese cara adiante.

Tras recoñecer unha compoñente léxica, inicio avanza ata a posición de dianteiro e iníciase a análise do seguinte lexema.

- Cada vez que dianteiro se move:
  1. Compróbase se se acadou o final do ficheiro (EOF).
  2. Compróbase se se acadou o final dun bloque. Funciona coma unha lista circular: se se sobrepasa o bloque A, lese B; e se se sobrepasa B, lese A. Nesas operacións solicítaselle ao SO a carga doutro bloque.

Este método ten dúas **limitacións**:

→ O **tamaño do lexema está limitado por n**. Non é un problema importante.

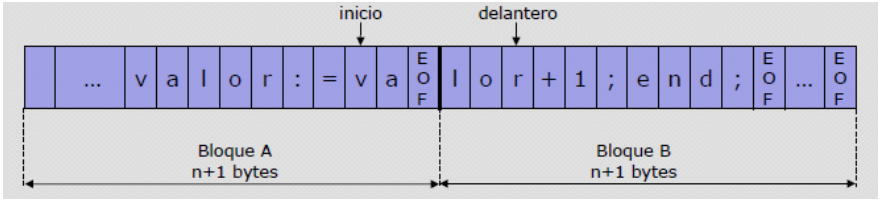
→ É **pouco eficiente**. Cada vez que dianteiro avanza hai que avaliar tres expresións lóxicas: fin de ficheiro, fin de bloque A e fin de bloque B.

O SISTEMA DE ENTRADA

método do centinela

Este método mellora o anterior, engadindo un byte máis a cada bloque no que se gardará un **caracter centinela** (EOF).

Deste xeito, o faise unha soa comprobación lóxica cada vez que avanza dianteiro: a de EOF. Se a comprobación é positiva, analízase cal dos tres casos é.



A TÁBOA DE SÍMBOLOS

A **táboa de símbolos** é a estrutura de datos utilizada polo compilador para **xestionar os identificadores** que aparecen no programa fonte: constantes, variables, tipos, funcións, ...

Cando o compilador atopa un identificador, garda nesta táboa a información que o caracteriza: nome, **categoría** (subrutina, variable, constante, clase, tipo, ...), a dirección de memoria que se lle asigna, o seu tamaño, etc. Cando o identificador é referenciado polo programa, o compilador **consulta** a táboa de símbolos e obtén a información que necesita. Unha vez fóra do **ámbito** do identificador, elimínase da táboa de símbolos.

Pazo

A TÁBOA DE SÍMBOLOS

venres, 9 de febreiro de 2024 12:25

cómo se emprega durante a compilación?

- O **analizador léxico**, cando atopa un identificador, comproba que está na lista de símbolos. Se non o está, crea unha nova entrada para o mesmo.
- O **analizador sintáctico** engade información aos campos dos atributos, pero tamén pode crear novas entradas se se definen novos tipos de datos como palabras reservadas.
- A **análise semántica** debe acceder á táboa para consultar os tipos de datos dos símbolos.
- O **xerador de código** pode:
  1) Ler o tipo de datos dunha variable para a reserva de espazo.
  2) Gardar a dirección de memoria na que se almacenará unha variable.

A TÁBOA DE SÍMBOLOS

palabras reservadas

Algunhas linguaxes reservan algunhas palabras que non poden empregarse como identificadores: `printf`, `for`, `if`, `while`, etc.

Cómo as distinguimos? 3 formas:

1) Podemos definilas mediante **expresións regulares** e asociarlles unha compoñente léxica particular.
   `[Ww][Hh][Ii][Ll][Ee]` `return(_WHILE)`
2) Mediante unha **táboa de palabras clave** na que se busque cada vez que se atope un lexema correspondente a un identificador.
3) Insertándoas ao principio da **táboa de símbolos**.

A táboa de símbolos inicialízase nas primeiras posicións coas palabras reservadas da linguaxe, ordenadas alfabeticamente.

Cando se atopa un nome no código fonte consúltase a táboa de símbolos:

→ Se se atopa entre as palabras reservadas, devólveselle a súa compoñente léxica ao analizador sintáctico.

→ Se se atopa despois das palabras reservadas é un identificador previamente atopado.

→ Se non se atopa na táboa de símbolos, engádese como un novo identificador.

Deste xeito, **redúcese o tamaño** do AFD e **auméntase a eficiencia** da análise léxica.

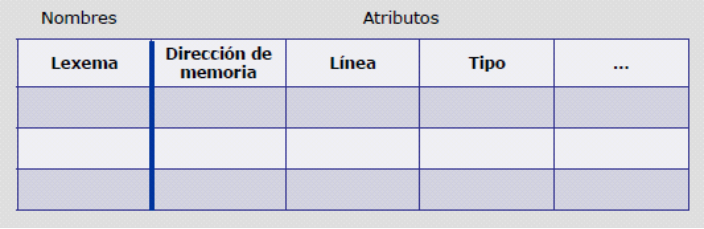
A TÁBOA DE SÍMBOLOS

a estrutura da táboa

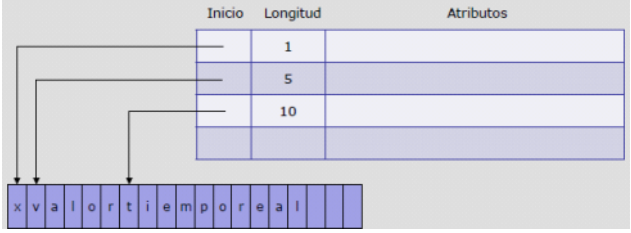
A táboa de símbolos estrutúrase nun **conxunto de rexistros** -de lonxitude usualmente fixa-, contendo o lexema atopado e un conxunto de atributos para a súa compoñente léxica.

Hai dúas formas características de almacenamento:

1. Estrutura interna:



2. Estrutura externa:



Esta estrutura non esixe unha lonxitude fixa para os identificadores, o que permite aproveitar mellor o espazo de almacenamento.

Pazo

A TÁBOA DE SÍMBOLOS

venres, 9 de febreiro de 2024 13:00

operacións

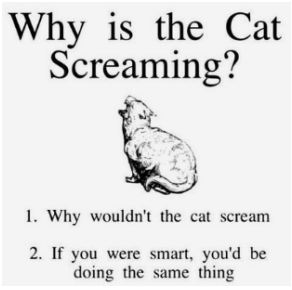
A táboa de símbolos funciona como unha **base de datos** na que o campo clave é o lexema do símbolo.

As operacións que se executan sobre a táboa son:

- **Buscar** un lexema e o contido dos seus atributos.
- **Insertar** un novo rexistro previa comprobación da súa non existencia.
- **Modificar** a información contida nun rexistro. En xeral, realízase unha operación de adición de información.

Ademais, en linguaxes con estrutura de bloque:

- **Novo bloque**: comezo dun novo bloque.
- **Fin de bloque**: final dun bloque.



A TÁBOA DE SÍMBOLOS

organización da táboa

As táboas de símbolos organízanse de dúas maneiras:

- Táboas non ordenadas. Xeradas con vectores ou listas. Pouco eficientes pero doadas de programar.
- Táboas ordenadas. Permiten definir o tipo dicionario. Empregan estruturas de tipo vector ou lista ordenada, árbores binarias, árbores equilibradas (AVL), táboas de dispersión (*hash*), etc.

Analizaremos o uso dalgunhas destas estruturas para organizar unha táboa de símbolos. Ocuparémonos do seu uso na tradución de linguaxes con estrutura de bloques.

As **regras de ámbito**, propias de cada linguaxe, determinan a definición de cada lexema en cada momento da compilación.

Exemplo:

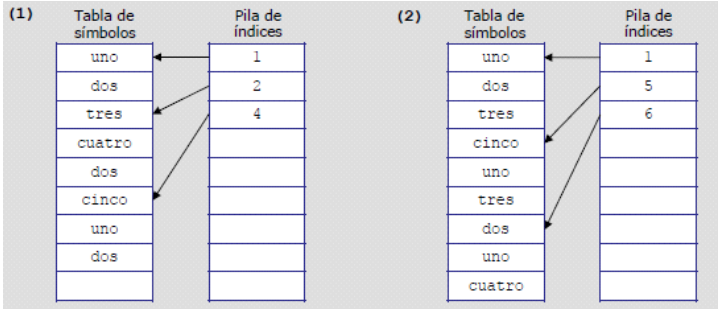
Programa	Nivel	Bloque
`program uno`	1	1
`  var dos: integer;`	2	
`  procedure tres`	2	2
`    var cuatro: char;`	3	
`    procedure dos`	3	3
`      var cuatro: integer;`	4	
`    procedure cinco`	3	4
`      var uno,dos: real;  (1)`	4	
`  procedure cinco`	2	5
`    var uno,tres: real;`	3	
`    procedure dos`	3	6
`      var uno,cuatro:char;  (2)`	4	

A TÁBOA DE SÍMBOLOS

táboas de símbolos non ordenadas

Emprégase un vector ou unha lista. Engádese unha **pila auxiliar de apuntadores de índice de bloque**, para marcar o comezo dos símbolos que lle corresponden a un bloque.

Ao rematar un bloque, elimínanse todos os símbolos dende o seguinte ao apuntado ata o final da táboa de símbolos.



Pazo

A TÁBOA DE SÍMBOLOS

xoves, 15 de febreiro de 2024 09:13

táboas de símbolos non ordenadas

Esta organización admite as seguintes operacións:

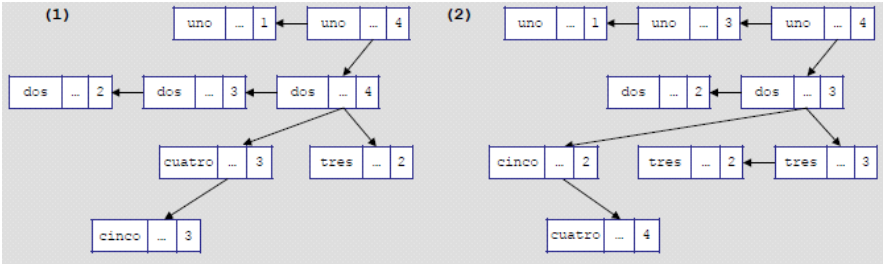
- **Inserción**. Cando se atopa a declaración dun símbolo débese verificar que non se atopa no último bloque. Se non está, insértase na última posición da táboa; e se está devólvese un erro ("símbolo xa definido nese ámbito").
- **Busca**. A busca faise dende o final da táboa cara ao comezo. Se se atopa, correspóndese á declaración realizada no bloque máis próximo.
- **Novo bloque**. Cando comeza un novo bloque, engádese na pila un apuntador ao último símbolo da táboa.
- **Fin de bloque**. Cando remata un bloque elimínanse todos os símbolos de dito bloque dende o seguinte ao de inicio, apuntado dende a pila. Despois, elimínase o índice de pila.

A TÁBOA DE SÍMBOLOS

táboas de símbolos con estrutura de árbore

Mellórase a eficiencia da táboa de símbolos mediante unha **árbore binaria ordenada**, e engádeselle un novo campo a cada rexistro que indique o nivel ao que pertence o símbolo.

Para evitar duplicidades utilízanse listas encadeadas.



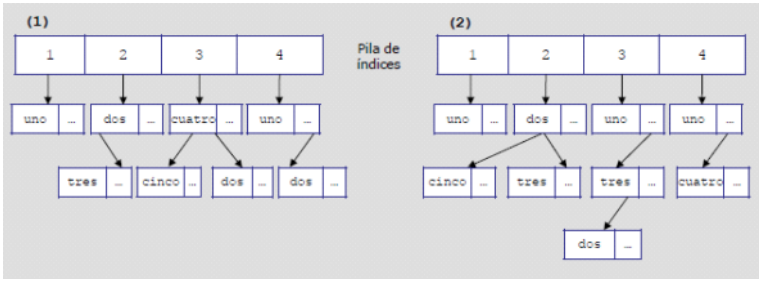
Esta organización admite as seguintes operacións:

- **Inserción**. Para insertar un novo símbolo, primeiro búscase a posición que lle corresponde. Se xa hai un rexistro co mesmo lexema, compróbase que o campo de nivel non ten o mesmo valor ca o bloque activo. Se fose así indícase erro. Se non, engádeselle o novo nodo á lista do primeiro. Se non hai rexistro insértase na ábrore.
- **Busca**. A propia dunha árbore binaria.
- **Novo bloque**. Increméntase en 1 o campo de nivel.
- **Fin de bloque**. Ao rematar un bloque, elimínanse os seus símbolos locais. Para isto:
  1. Localízanse os rexistros da árbore do bloque activo.
  2. Bórranse.
  3. Decreméntase en 1 a variable co número de nivel.

A TÁBOA DE SÍMBOLOS

táboas de símbolos con estrutura de bosque

Esta técnica emprega **unha árbore para cada bloque do programa**, e unha pila de índices de nivel que apuntan á raíz da árbore de nivel.



Pazo

A TÁBOA DE SÍMBOLOS

xoves, 15 de febreiro de 2024 09:30

táboas de símbolos con estrutura de bosque

Esta organización admite as seguintes operacións:

- **Inserción**. Consiste en facer unha inserción normal na árbore do bloque activo.
- **Busca**. Primeiro, búscase o lexema na árbore do bloque activo. Se non se atopa, búscase na árbore do bloque anterior, e así sucesivamente.
- **Novo bloque**. Cando comeza a compilación dun novo nivel, créase un novo elemento na pila de índices e unha nova estrutura de tipo árbore.
- **Fin de bloque**. Cando se termina de compilar un bloque, destrúese a árbore asociada e elimínase o último punteiro da pila de índices.

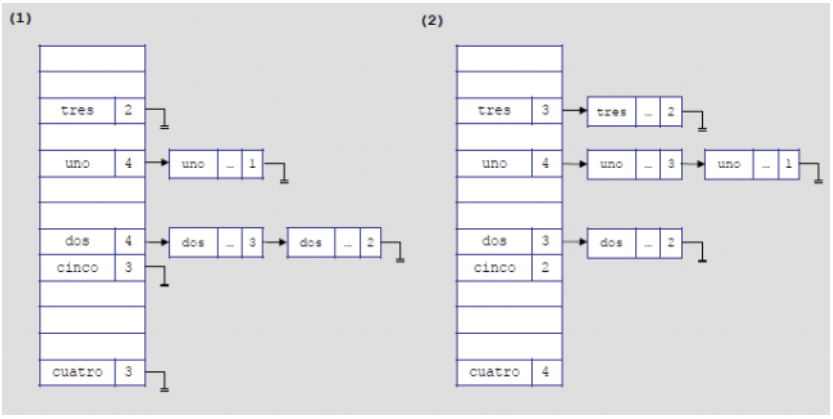
A TÁBOA DE SÍMBOLOS

táboas de dispersión (*hash*)

Unha táboa de dispersión aplícalle unha **función matemática** ao lexema para determinar a posición da táboa que lle corresponde.

O inconvinte das táboas de dispersión son as **colisións** que se producen cando dous lexemas queren ocupar a mesma posición. Existen dúas solucións:

1) **Táboas de dispersión pechadas**. Cando se produce unha colisión, emprégase algunha técnica que permita acceder a unha posición baleira (son lineal, multiplicativo, cadrático, ...).
2) **Táboas de dispersión abertas**. Emprégase unha lista encadeada de desbordamento para resolver as colisións. Cando se produce unha colisión, créase unha lista encadeada coa cabeza no rexistro da táboa.



Esta organización admite as seguintes operacións:

- **Inserción**. Coma en calquera táboa de dispersión. Adóitanse empregar listas de desbordamento, colocando os símbolos máis recentes nas primeiras posicións.
- **Busca**. Coma en calquera táboa de dispersión. En caso de colisión, débese buscar na zona de desbordamento, ata atopar o símbolo ou unha posición baleira.
- **Novo bloque**. Só hai que almacenar o cambio de bloque activo.
- **Fin de bloque**. Cando se termina de compilar un bloque, borraranse todos os rexistros correspondentes a el. Para isto, percórrese practicamente toda a táboa. Despois, cámbiase de bloque activo.

ANÁLISE DE COMPLEXIDADE

	Búsqueda	Inserción
Lista no ordenada	O(n)	O(n)
Lista ordenada	O(log n)	O(n)
Árbol AVL	O(log n)	O(log n)
Tabla de dispersión	O(1)	O(1)

Pazo

TRATAMENTO DE ERROS

xoves, 15 de febreiro de 2024 12:17

Cando nun momento do proceso de compilación se detecta un erro:

- Débese mostrar unha mensaxe **clara e exacta**, que lle permita ao programador atopalo e corrixilo facilmente.
- Débese **recuperar do erro** e ir a un estado que permita continuar analizando o programa en busca de outros erros. Débese evitar unha cascada de erros ou pasar por alto outros.
- **Non debe retrasar** excesivamente o procesamento de programas correctos.

Os erros máis característicos da fase da análise léxica son:

Tipo de error	Explicación y subtipos	Ejemplos
Nombres ilegales de identificadores	Nombre de identificador que contiene caracteres inválidos	nen@12'
Números inválidos	Contiene caracteres inválidos Está formado incorrectamente Es demasiado grande y produce desbordamiento	2:13 3.14.58 9999999999999
Cadenas de caracteres incorrectas	Cadena demasiado larga, probablemente porque falta cerrar unas comillas	
Errores de ortografía en palabras reservadas	Caracteres omitidos Caracteres adicionales Caracteres incorrectos Caracteres mezclados	wile whhile whule wihle
Fin de fichero	Se detecta un fin de fichero durante el análisis de un componente léxico	

O proceso de recuperación dun erro pode supor a adopción de diferentes medidas:

- **Ignorar** os caracteres inválidos ata formar unha compoñente léxica correcta.
- **Eliminar** caracteres que dan lugar a erro.
- Intentar corrixir o erro:
  - **Insertar** os caracteres que poidan faltar.
  - **Reemplazar** un carácter presuntamente incorrecto por outro correcto.
  - **Intercambiar** caracteres adxacentes.
- *Coidado: intentar corrixir un erro pode ser perigoso.*

Pazo

INTRODUCIÓN Á ANÁLISE SINTÁCTICA

xoves, 15 de febreiro de 2024 12:33

Neste tema estudiaremos a segunda fase do proceso de compilación: a **análise sintáctica** do programa fonte.

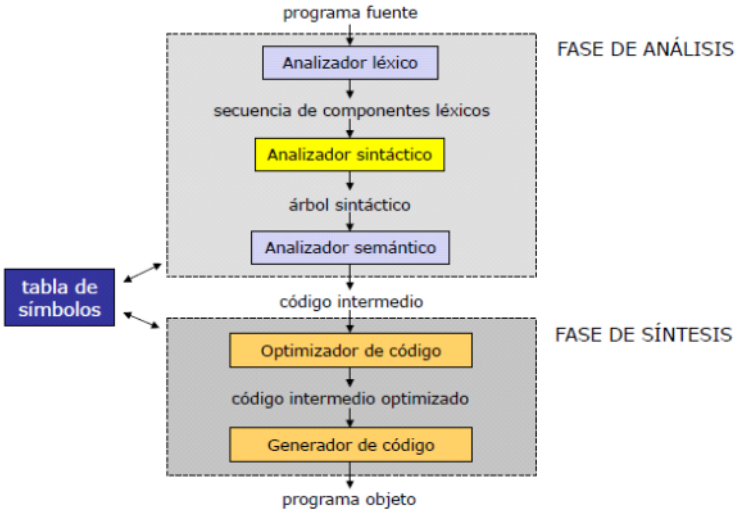
Unha **gramática** é a ferramenta formal que nos vai axudar a comprobar a estrutura dun programa.

Dividiremos o tema en tres bloques:

- No primeiro bloque, daremos unha visión xeral do analizador sintáctico, e algunhas das cuestións fundamentais á hora de deseñar gramáticas: **recursividade**, **ambigüidade**, **asociatividade** e **precedencia**.
- No segundo bloque, estudaremos a **análise sintáctica descendente**.
- No terceiro bloque, estudaremos a **análise sintáctica ascendente**.

! Vai haber unha pregunta de cada un dos dous últimos bloques no exame, que ademais serán as de máis peso. *"Con elas acertadas xa se ten aprobado o final."*

ESTRUTURA



NOCIÓNS XERAIS

gramáticas independentes do contexto

Unha **gramática independente do contexto** defínese como unha 4-tupla G=(Σ, V, S, P), sendo:

- Σ un alfabeto ou conxunto de **símbolos terminais**.
- V un conxunto de variables ou **símbolos non terminais**.
- S unha variable denominada **símbolo inicial**.
- P unha colección de **regras de substitución**, chamadas regras de produción, da forma $A \to \alpha$, onde $A \in V$ e $\alpha \in (V \cup \Sigma)^*$.

A máquina apropiada para o recoñecemento de linguaxes independentes de contexto é o **autómata de pila**.

- Neste tema introduciremos o símbolo $ para indicar o **fin de cadea**.

Pazo

NOCIÓNS XERAIS

xoves, 15 de febreiro de 2024 21:06

tipos de analizadores sintácticos

Un **analizador sintáctico** analiza o código fonte coma unha secuencia de compoñentes léxicas (símbolos terminais), e constrúe unha representación interna en forma de árbore sintáctica. Podemos falar de dous tipos:

1. **Descendentes**: parten da raíz da árbore. Aplican as regras da gramática, substituíndo a parte esquerda dunha regra pola súa parte dereita, e xerando a árbore sintáctica de arriba á abaixo.
2. **Ascendentes**: a árbore constrúese de abaixo a arriba; das follas á raíz. Para isto, aplica as regras da gramática reducindo a parte dereita dunha produción pola súa parte esquerda, subindo poolos nodos ata chegar á raíz: o símbolo inicial S.

NOCIÓNS XERAIS

forma de Backus-Naur (BNF) extendida

É a notación máis común para **formalizar** gramáticas. Os metasímbolos máis comúns que emprega son:

[NOTACIÓN BNF BÁSICA]

- ::=, empregada en definicións.
- |, empregada para representar disxunción.
- <>, empregados para representar símbolos non terminais.

[NOTACIÓN BNF EXTENDIDA]

- [], empregados para representar símbolos opcionais, é dicir, que poidan aparecer unha vez ou ningunha.
- {}, empregados para representar repetición: cero, unha ou máis veces.
- ", empregadas para distinguir metasímbolos de terminais.

NOCIÓNS XERAIS

forma de Backus-Naur (BNF) visual

- Os símbolos terminais márcanse en negrita.
- Elimínanse os símbolos <>.
- Emprégase o símbolo → no lugar de ::=.

Exemplo:

sentencia_if → ***if*** *expresion* ***then***
sentencia
[***else***
sentencia }
endif*;*

DESEÑO DUNHA GRAMÁTICA

Cando se desenvolve unha nova linguaxe de programación, débese deseñar unha **nova gramática** que describirá as características da linguaxe. Algúns aspectos que se deben discutir son:

1. **Recursividade**. O que permite reducir o número de regras sintácticas.
2. **Ambigüidade**. Débese evitar: proporciona unha gramática usualmente máis intuitiva, pero permite xerar diferentes códigos obxecto para o mesmo código fonte.
3. **Asociatividade**. Débense proporcionar regras de asociatividade para as distintas expresións da linguaxe.
4. **Precedencia**. Determina a orde na que se van realizar as distintas operacións da linguaxe.

Pazo

DESEÑO DUNHA GRAMÁTICA

sábado, 17 de febreiro de 2024 19:05

recursividade

As linguaxes de programación permiten xerar un número ilimitado de programas, o que obriga a empregar un método de xeración sen realizar unha casuística interminable. A **recursividade** permite xerar un número infinito de programas mediante unha **gramática finita**.

→ Unha gramática é **recursiva** se, ao reescribir un non terminal, este volve a aparecer nunha ou máis derivacións $A \Rightarrow^{+} \alpha A \beta$

NOTA IMPORTANTE!!!!!
Esta reaparición pode suceder dentro do primeiro nivel ou cando xa levamos unha chea de substitucións. Vamos, que non ten por que ser en substitucións consecutivas.

Exemplo:

```
<sentencia> ::= begin <sentencia> {; <sentencia>} end
<sentencia> ::= <asignacion> | <chamada_funcion> |
                        <sentencia_repetitiva> |
                        <sentencia_condicional>
```

DESEÑO DUNHA GRAMÁTICA

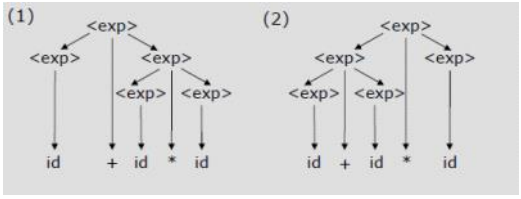
ambigüidade

→ Unha gramática é **ambigua** se xera, polo menos, unha cadea mediante dúas ou máis árbores de derivación diferentes.

Débese evitar a ambigüidade, xa que podendo utilizar distintas árbores sintácticas para xerar unha mesma sentencia, **o código xerado poderá ser distinto** en distintas execucións do compilador.

Exemplo:

```
<expresion> ::= <expresion> + <expresion> |
                <expresion> * <expresion> |
               (<expresion>) | - <expresion> | id
```



DESEÑO DUNHA GRAMÁTICA

factorización pola esquerda

En ocasións, durante o deseño dunha gramática, aparecen **dúas producións dun mesmo non terminal que empezan igual**. Cando durante a análise se chegue a este non terminal, non se saberá cál delas escoller.

Exemplo:

```
<sentencia> ::= if <expresion> then <sentencia> |
                if <expresion> then <sentencia>
                                    else <sentencia>
```

Unha solución é **reescribir** estas regras, retrasando a decisión ata ter visto abondo. Chamámoslle a esta solución **factorización pola esquerda**.

En xeral, se a nosa gramática ten unha produción da forma $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma$, podemos realizar a substitución seguinte:

$$A \rightarrow \alpha A' \mid \gamma$$
$$A' \rightarrow \beta_1 \mid \beta_2 \mid \ldots \beta_n$$

e avaliar se $\beta_1 \mid \beta_2 \mid \ldots \beta_n$ tamén se pode factorizar.

DESEÑO DUNHA GRAMÁTICA

sábado, 17 de febreiro de 2024 21:49

asociatividade

Cando deseñamos unha linguaxe debemos decidir o tipo de asociatividade das operacións entre máis de dous operandos:

1. **Asociatividade pola dereita**. As operacións fanse de dereita a esquerda. Por exemplo, a asignación.

$$a = b = c \equiv a = (b = c)$$

2. **Asociatividade pola esquerda**. As operacións fanse de esquerda a dereita. Por exemplo, as aritméticas.

$$a + b + c \equiv (a + b) + c$$

A asociatividade pódeselle incorporar a unha gramática mediante a **recursividade**: distinguimos entre operador asociativo pola dereita e pola esquerda:

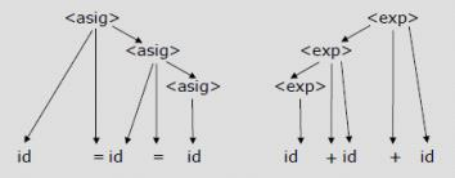
1. **Operador asociativo pola dereita**. A regra sintáctica na que aparece o operador faise recursiva pola dereita.

```
<asignacion> ::= id = <asignacion> | id
```

2. **Operador asociativo pola esquerda**. A regra sintáctica na que aparece o operador faise recursiva pola esquerda.

```
<expresion> ::= <expresion> + id | id
```

Exemplo:



DESEÑO DUNHA GRAMÁTICA

precedencia

Cando nunha expresión interveñen varios operadores pódese producir ambigüidade ao construír a árbore de derivación. A **precedencia** permite especificar unha **orde relativa de avaliación** duns operadores respecto a outros. Pódese incorporar empregando:

a. Unha variable sintáctica para cada nivel de precedencia.
b. Unha produción para cada operador.

→ Un operador terá menor precedencia canto máis preto estea da súa regra de derivación da regra inicial da gramática.

Exemplo:
Desexamos unha gramática asociativa pola esquerda, coa suma e a resta con precedencia 1, a multiplicación e a división con precedencia 2, a potenciación con precedencia 3 asociada pola dereita, e o paréntese con precedencia máxima.

```
<expresion> ::= <expresion> + <expr_mult> |
                <expresion> - <expr_mult> |
                <expr_mult>
<expr_mult> ::= <expr_mult> * <expr_esp> |
                <expr_mult> / <expr_esp> |
                <expr_esp>
<expr_esp> ::= <valor> * <expr_esp> | <valor>
<valor> ::= (<expresion>) | id
```

Pazo

ANÁLISE SINTÁCTICA DESCENDENTE (ASD)

sábado, 17 de febreiro de 2024 22:51

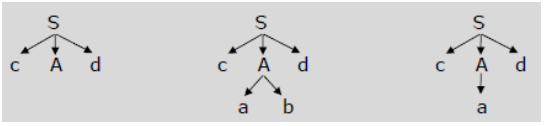
Unha vez definida unha gramática, debemos comprobar que cada programa escrito en código fonte lle obedece ás regras da gramática. Este proceso é a **análise sintáctica**.

O caso máis sinxelo de ASD é totalmente recursivo:

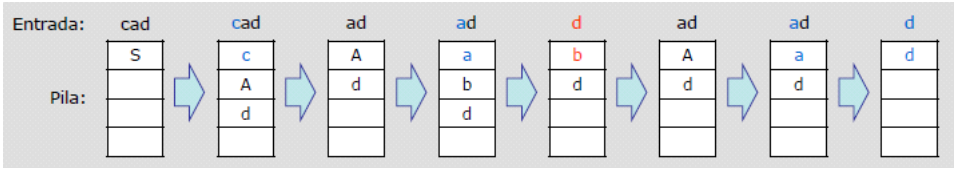
1) **Avanzamos**: cando substituímos un non terminal por algunha das súas producións.
2) **Retrocedemos**: cando se chega a un punto morto e é necesario probar outra regra de substitución.

Para xestionar este proceso empregamos unha pila, onde almacenamos os símbolos de cada substitución. Intentaremos emparellar o símbolo que hai no cumio da pila coa entrada actual.

Exemplo: queremos saber se a entrada $w = cad$ pertence á gramática seguinte: $S \to cAd$, $A \to ab \mid a$



Se ao final do proceso, a pila e a entrada están baleiras entón a cadea pertence á linguaxe.



Unha posible implementación do ASD pódese realizar desenvolvendo unha función de análise por cada non terminal.

Estes métodos de ASD teñen algúns inconvintes que non os fan moi recomendables:

1. Son moi **lentos** debido aos retrocesos.
2. Non se sabe se a entrada pertence ou non á linguaxe ata **analizar todas as posibilidades**. Se non pertence, na pila temos o símbolo S. Non podemos dicir onde está o erro.
3. Se se xera código obxecto durante a análise, en cada retroceso hai que **eliminar parte do código xerado**.

ANÁLISE SINTÁCTICA DESCENDENTE (ASD)

recursividade pola esquerda

As regras recursivas pola esquerda poden facer entrar ao ASD nun **bucle infinito**. Debemos, polo tanto, eliminar a recursividade pola esquerda.

Sexa unha gramática recursiva:

$$A \to A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_m$$

Unha gramática equivalente non recursiva pola esquerda será:

$$A \to \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_m A'$$
$$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_n A' \mid \varepsilon$$

Exemplo:
$S \to cAd$
$A \to Aad \mid a$
⇨ $\alpha = ad, \beta = \alpha$ ⇨
$S \to cAd$
$A \to aA'$
$A' \to adA' \mid \varepsilon$

Pazo

ANÁLISE SINTÁCTICA DESCENDENTE (ASD)

xoves, 22 de febreiro de 2024 09:58

recursividade pola esquerda

O método anterior elimina a recursividade inmediata -é dicir, na mesma derivación-. Se a recursividade aparece en derivacións posteriores debemos atopar o elemento conflitivo e substituílo pola súa definición.

Exemplo:

$S \to Aa \mid b$
$A \to Ac \mid Sd \mid \varepsilon$

⇨

$S \to Aa \mid b$
$A \to Ac \mid Aad \mid bd \mid \varepsilon$

Existe recursividade pola esquerda na aplicación das regras de substitución $S \to Aa$ e $A \to Sd$.
Substituímos $Sd \to Aad \mid bd$.
E aplicamos o método anterior:

$S \to Aa \mid b$
$A \to bdA' \mid A'$
$A' \to cA' \mid adA' \mid \varepsilon$

⇦ $\alpha = c \mid ad, \beta = bd \mid \varepsilon$

ANÁLISE SINTÁCTICA DESCENDENTE (ASD)

analizador descendente preditivo

Desexamos evitar os retrocesos, e **predicir** en cada momento cál das regras debemos aplicar para continuar a análise correctamente.

- Supoñamos que a cadea de entrada é $c_1 \dots c_i \dots c_n$.
- Supoñamos que a compoñente léxica actual é c_i.
- Supoñamos que o non terminal a substituír é A.
- As substitucións posibles son $A \to \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$.

Se cada unha das regras de substitución comeza por unha compoñente léxica distinta debemos aplicar a regra de substitución que comeza por c_i.

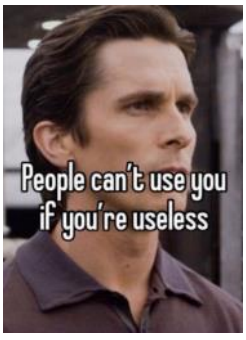
O tipo de gramáticas que se propoñen son as **LL(k)**:
- L (*left*): a entrada lese de esquerda a dereita.
- L (*left*): substitúese o non terminal situado á esquerda.
- (k): lense k compoñentes léxicas por anticipado.

Centrarémonos en gramáticas **LL(1)**, cunha soa compoñente léxica analizada por anticipado.
Unha gramática LL(1) debe cumprir:

1. Non pode ser recursiva pola esquerda.
2. Dúas regras se substitución dun mesmo terminal non poden dar lugar á mesma compoñente léxica inicial.

Se atopamos unha gramática non LL(1) modificarémola **eliminando a recursividade** pola esquerda e sacando **factor común** pola esquerda.

! Estas son condicións necesarias, pero NON suficientes.



Pazo

ANÁLISE SINTÁCTICA DESCENDENTE (ASD)

xoves, 22 de febreiro de 2024 10:47

conxuntos de predición

Definimos dous conxuntos de predición, que axudarán no proceso da análise preditiva: o conxunto **PRIMEIROS** e o conxunto **SEGUINTES**.

CONXUNTO PRIMEIROS

Sexa X un símbolo gramatical ($X \in (V \cup \Sigma)$), PRIMEIROS(X) é o conxunto de terminais (incluíndo ε) que aparecen ao inicio das cadeas derivables de X en ningún ou máis pasos.

$$PRIMEIROS(X) = \{v \mid X \Rightarrow^* v\beta, v \in \Sigma, \beta \in \Sigma^*\}$$

Para calcular PRIMEIROS(X):

1) Se X é un símbolo terminal a, PRIMEIROS(X) = a.
2) Se (X → ε) ∈ P, engadimos ε a PRIMEIROS(X).
3) Se X é un non terminal, e $X \rightarrow Y_1 Y_2 \dots Y_k$ é unha regra de substitución, engadimos PRIMEIROS(Y_j) a PRIMEIROS(X), se PRIMEIROS(Y_j) é un terminal, e para todo i=1, 2, ..., j-1, $Y_i \Rightarrow^* \varepsilon$.
4) Se X é un non terminal, $X \rightarrow Y_1 Y_2 \dots Y_k$ é unha regra de substitución, e $Y_j \Rightarrow^* \varepsilon \; \forall \; j \in \{1, 2, \dots, k\}$, engadimos ε a PRIMEIROS(X).

Para calcular PRIMEIROS($X_1 X_2 \dots X_n$):

1) Engadimos todos os símbolos distintos de ε de PRIMEIROS(X_1).
2) Se ε está en PRIMEIROS(X_1), engadir tamén os símbolos distintos de ε de PRIMEIROS(X_2).
3) Se ε está en PRIMEIROS(X_2), engadir tamén os símbolos distintos de ε de PRIMEIROS(X_3). E así sucesivamente.
4) Por último, engadir ε a PRIMEIROS($X_1 X_2 \dots X_n$) se PRIMEIROS(X_i) contén a ε, $\forall \; i \in \{1, 2, \dots, n\}$.

CONXUNTO SEGUINTES

Sexa A un símbolo non terminal. SEGUINTES(A) é o conxunto de terminais que poden aparecer inmediatamente á dereita de A nalgunha substitución.

$$SEGUINTES(A) = \{v \mid S \Rightarrow^+ \alpha A v \beta, v \in \Sigma; \; a, \beta \in \Sigma^*\}$$

Se A aparece ao final dunha cadea de símbolos, entón diremos que o símbolo de fin de cadea $ ∈ SEGUINTES(A).

Para calcular SEGUINTES(A):

1) Se A é S, engadir $ a SEGUINTES(S).
2) Para cada regra $B \rightarrow \alpha A \beta$, engadir PRIMEIROS(β)-ε a SEGUINTES(A).
3) Para cada regra $B \rightarrow \alpha A \beta$, onde PRIMEIROS(β) contén a ε (β ⇒* ε), engadir todos os símbolos de SEGUINTES(B) A SEGUINTES(A).
4) Para cada regra $B \rightarrow \alpha A$, engadir SEGUINTES(B) a SEGUINTES(A).

Pazo

ANÁLISE SINTÁCTICA DESCENDENTE (ASD)

xoves, 22 de febreiro de 2024 14:46

táboa de análise sintáctica

Mediante os conxuntos de predición podemos construír unha táboa que, a partir do non terminal a expandir e da compoñente léxica actual, indique que regra se debe aplicar.

Nesta **táboa**, as **filas** son os **símbolos non terminais** da gramática, e as **columnas** son as **compoñentes léxicas** e o símbolo de fin de cadea **$**. Nas celas da táboa aparecen regras da gramática. As celas baleiras correspóndense cun erro na gramática.

As celas da táboa T[A, a] énchense seguindo o seguinte procedemento para cada regra de substitución $A \to a$:

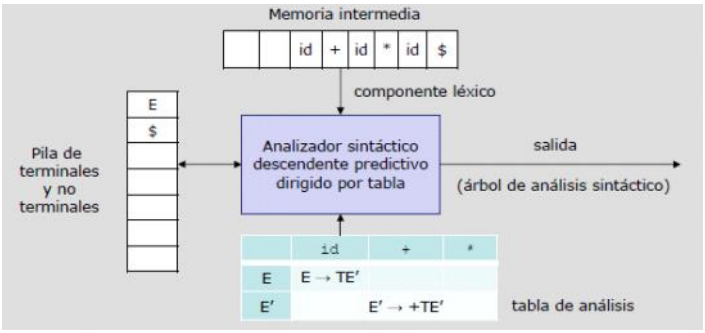
1) Para cada terminal $a \in PRIMEIROS(a)$, engadir $A \to a$ en T[A, a]
2) Se $\varepsilon \in PRIMEIROS(a)$ para cada terminal $b \in SEGUINTES(A)$, engadir $A \to a$ en T[A, a].

Unha gramática é LL(1) se na táboa de análise sintáctica aparece **como máximo unha regra en cada cela**.

ANÁLISE SINTÁCTICA DESCENDENTE (ASD)

procedemento de análise

Podemos proporcionar un procedemento de análise baseada no uso dunha táboa de análise sintáctica para seleccionar a seguinte táboa de substitución e unha pila para almacenar os símbolos gramaticais.



O algoritmo de análise é o seguinte:

1) Inicializar a pila co símbolo S, e engadir $ a continuación e ao final da cadea de entrada.
2) Comparar o símbolo X do cumio da pila e o seguinte símbolo *a* da entrada:
   i) Se X = a = $, **aceptar** a cadea e saír.
   ii) Se X = a ≠ $, extraer o elemento da pila e avanzar unha posición na cadea de entrada.
   iii) Se X é un terminal distinto de a, ou X é un non terminal e T(X, a) está baleira, entón **erro**.
   iv) Se X é un non terminal e en T(X, a) temos $X \to X_1X_2 \dots X_n$, extraer X da pila e insertar $X_1X_2 \dots X_n$.
   v) Volver a 2.

ANÁLISE SINTÁCTICA DESCENDENTE (ASD)

<<recapitulemos>>

As gramáticas LL(!) admiten un proceso de análise cunha **complexidade O(n)**. Non todas as gramáticas son LL(1), pero nalgúns casos poden levarse a cabo certas transformacións para convertelas en LL(1):

1) **Eliminar a recursividade pola esquerda**: se a gramática é recursiva pola esquerda, durante a análise entraremos nun bucle infinito. Vimos regras para eliminala.
2) **Eliminar a ambigüidade**: se a gramática é ambigua non é LL(1), xa que ante un símbolo de entrada pode xerarse máis dunha árbore sintáctica. Debemos rediseñar a gramática.
3) **Factorizar pola esquerda**: se dúas substitucións empezan polo mesmo terminal non saberemos cál elexir. A factorización pola esquerda permite retrasar a decisión ata que a entrada nos permita tomar a correcta.

Pazo

ANÁLISE SINTÁCTICA DESCENDENTE (ASD)

xoves, 29 de febreiro de 2024 09:40

xestión de erros

Poden darse dous tipos de erros no ASD:

1. Na parte superior da pila hai un **terminal que non se corresponde coa compoñente léxica actual**. O compilador deberá sinalar o erro, indicando que é o que esperaba atopar.
2. Na parte superior da pila hai un **non terminal**, e a **cela** da táboa correspondente á compoñente léxica actual está **baleira**. O compilador deberá sinalar un erro, conforme se esperaba unha das compoñentes léxicas do conxunto de celas do non terminal que conteñen algún dato.

Ao detectar o erro, o compilador debe continuar a análise. Para facelo, pode aplicar a técnica de **conxuntos de sincronización**.

ANÁLISE SINTÁCTICA DESCENDENTE (ASD)

conxuntos de sincronización

Esta técnica baséase en que, ao detectar un erro, se entra nun **estado de erro** no que se continúa analizando a entrada, descartando as compoñentes léxicas que aparecen ata atopar unha que pertenza ao **conxunto de sincronización**.

Nese momento, valórase quitar o símbolo do cumio da pila, abandonar o estado de erro e proseguir a análise.

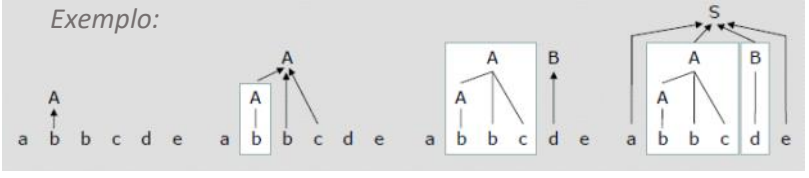
A efectividade deste método depende da elección do conxunto de símbolos de sincronización. Normalmente, empréganse **técnicas empíricas**, baseadas no estudo dos erros máis frecuentes.

ANÁLISE SINTÁCTICA ASCENDENTE (ASA)

A análise sintáctica ascendente parte dunha cadea de entrada formada por unha secuencia de compoñentes léxicas, e busca facela corresponder coas partes dereitas das regras da gramática.

Cando atopa unha, substitúea (dicimos que a reduce) pola súa parte esquerda, e crea un novo nodo da árbore sintáctica. Continúa así ata chegar ao símbolo inicial S ou atopar un erro.

Xera, polo tanto, a árbore sintáctica de abaixo a arriba: empeza polas follas, que se corresponden con compoñentes léxicas; e acaba na raíz, aplicando as regras de substitución en sentido inverso.



Pode haber varias regras de substitución con algunha parte dereita común, e regras que substitúan pola cadea baleira. O problema é decidir cándo o que parece unha parte dereita dunha regra se pode substituír pola súa parte esquerda.

Os algoritmos de ASA empregan unha pila na súa análise, e realizan dúas operacións básicas:

1. **Desprazamento**: leva o seguinte símbolo de entrada ao cumio da pila.
2. **Redución**: extrae tantos símbolos da pila como símbolos ten a parte dereita da regra de substitución pola que se quere reducir e substitúeos polo símbolo non terminal da parte esquerda da regra.

Pazo

ANÁLISE SINTÁCTICA ASCENDENTE (ASA)

xoves, 29 de febreiro de 2024 10:23

Exemplo:
*Analizar a cadea id+id*id coa gramática:*

$E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow (E)$
$E \rightarrow id$

Pila	Entrada	Acción
	id+id*id$	Desplazar
id	+id*id$	Reducir con E → id
E	+id*id$	Desplazar
E+	id*id$	Desplazar
E+id	*id$	Reducir con E → id
E+E	*id$	**Reducir con E → E+E, o desplazar**
E+E*	id$	Desplazar
E+E*id	$	Reducir con E → id
E+E*E	$	Reducir con E → E*E
E+E	$	Reducir con E → E+E
E	$	Aceptar

ANÁLISE SINTÁCTICA ASCENDENTE (ASA)
gramáticas LR(k)

Na análise ascendente por **desprazamento-redución** empréganse **gramáticas LR(k)**:

- **L (*left*)**: a entrada lese de esquerda a dereita.
- **R (*right*)**: constrúense derivacións pola dereita en orde inversa.
- **(k)**: lense k compoñentes léxicas por adiantado. Pódese demostrar que, para cada gramática LR(k) con k>1, hai unha gramática LR(1) equivalente. Por este motivo, normalmente trabállase con k=1.

As gramáticas LR(1) non son ambiguas e **permiten describir máis linguaxes ca as gramáticas LL(1)**, englobándoas.

Hai tres tipos de gramáticas LR, co mesmo esquema de control, baseado na construción e uso dunha táboa de análise. Pola maneira de construír esta táboa distinguiremos:

- **Gramáticas SLR(1)** (*Simple LR*): presentan a construción máis sinxela. Serán as que estudaremos neste curso.
- **Gramáticas LR(1)**: permiten describir un rango maior de gramáticas, pero a construción é máis complexa.
- **Gramáticas LALR(1)** (*Look Ahead LR*): supoñen un compromiso entre as dúas anteriores.

ANÁLISE SINTÁCTICA ASCENDENTE (ASA)
procedemento de análise

Cómo sabe un analizador sintáctico ascendente cándo desprazar e cándo reducir?

Empregaremos un procedemento baseado no deseño dun **autómata finito determinista**, que nos dirá se unha entrada pertence á linguaxe xerada pola gramática:

→ Os seus **estados** son conxuntos de elementos que representan situacións equivalentes na análise dunha entrada.
→ Unha función á que lle chamaremos *Ir_a* permitiranos simular as **transicións** do autómata para cada un dos símbolos da gramática.
→ Para unha gramática G creamos a súa **gramática aumentada**, resultado de engadir artificialmente a regra S'→S$. Cando vamos a reducir mediante esta nova regra, aceptamos.

Pazo

ANÁLISE SINTÁCTICA ASCENDENTE (ASA)

xoves, 29 de febreiro de 2024 10:42

gramáticas SLR(1): elementos

Un **elemento LR(0)**, ou elemento simplemente, é unha regra simple que contén unha **marca** nalgún lugar da parte dereita. Esta marca separa os símbolos da regra que foron recoñecidos dos pendentes de recoñecer.

Exemplo:

Elementos posibles	Símbolos reconocidos	Símbolos esperados
A → ·B-D	Ninguno	Primeros(B)
A → B·-D	B	-
A → B-·D	B-	Primeros(D)
A → B-D·	B-D	Siguientes(A)

Para as regras de substitución A→ ε só se obtén o elemento A→·.

ANÁLISE SINTÁCTICA ASCENDENTE (ASA)

gramáticas SLR(1): clausura()

A función **clausura()** aplícaselle a un conxunto de elementos I, e devolve un novo conxunto de elementos que son equivalentes respecto a onde chegou a análise. O conxunto de elementos que devolve clausura(I) constrúese da seguinte maneira:

1) Todo elemento de I engádese a clausura(I).
2) Se A→ $\alpha \cdot B\beta$ está en clausura(I) e B→ γ é unha regra da gramática, entón engadir B→· γ a clausura(I). Aplicar esta regra ata que non se poidan engadir novos elementos a clausura(I).

Clausura(I) agrupa a todos os elementos que describen o que se espera atopar a partir do momento actual da análise.

Así, se A→ $\alpha \cdot B\beta \in clausura(I)$, acabamos de ver na entrada unha cadea derivable de α, e esperamos atopar a continuación unha cadea derivable de $B\beta$, é dicir: unha daquelas cadeas resultado de substituír B de todas as formas posibles.

Por iso engadimos B→· γ a clausura(I). Se γ é unha cadea que comeza cun non terminal, repetimos este proceso.

Rematamos cando dispomos de todos os elementos que indiquen o principio da cadea esperada a partir do momento actual da análise.

Exemplo:
Obter os sucesivos conxuntos de clausura(I), onde I={E'→·E$}, para a seguinte gramática:

$$'E \rightarrow E\$$$
$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid id$$

I	Clausura(I) primer paso	Clausura(I) segundo paso	Clausura(I) tercer paso	Clausura(I) cuarto paso
				E' → ·E$
			E' → ·E$	E → ·E+T
		E' → ·E$	E → ·E+T	E → ·T
E' → ·E$	E' → ·E$	E → ·E+T	E → ·T	T → ·T*F
		E → ·T	T → ·T*F	T → ·F
			T → ·F	F → ·(E)
				F → ·id

Pazo

ANÁLISE SINTÁCTICA ASCENDENTE (ASA)

xoves, 29 de febreiro de 2024 11:10

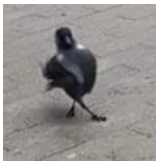
gramáticas SLR(1): ir_a(I, X)

A función **Ir_a(I, X)** aplícaselle a un **conxunto de elementos I** e a un **símbolo da gramática** $X \in (V \cup \Sigma)$; e devolve un conxunto de elementos novo.

- → Ir_a(I, X) defínese como a **clausura** do conxunto de todos os elementos A→ $\alpha X \cdot \beta$ tales que A→ $\alpha \cdot X\beta$ está en I.
- → Intuitivamente, a función Ir_a(I, X) **adianta** o símbolo X na análise sintáctica ascendente.
- → O conxunto que devolve Ir_a(I, X) está formado por aqueles elementos que despois de I **dan por recoñecido o símbolo X**.

Exemplo:

I_0=clausura(I)	I_1=Ir_a(I_0,E)	I_2=Ir_a(I_0,T)	I_3=Ir_a(I_0,F)	I_4=Ir_a(I_0,()	I_5=Ir_a(I_0,id)
E' → ·E$				F → (·E)	
E → ·E+T				E → ·E+T	
E → ·T	E' → E·$	E → T·		E → ·T	
T → ·T*F	E → E·+T	T → T·*F	T → F·	T → ·T*F	F → id·
T → ·F				T → ·F	
F → ·(E)				F → ·(E)	
F → ·id				F → ·id	



ANÁLISE SINTÁCTICA ASCENDENTE (ASA)

colección canónica LR(0)

Chamámoslle **colección canónica LR(0)** ao conxunto de todos os posibles conxuntos de elementos dunha gramática. Para xerala:

1) Comezamos creando a **gramática aumentada**, resultado de engadir artificialmente a regra $S' \rightarrow S\$$, da gramática da que queiramos xerar a colección canónica.
2) O primeiro conxunto de elementos é $I_0 = clausura(S' \rightarrow S\$)$.
3) Calcúlanse todos os posibles conxuntos resultado de aplicar Ir_a(I_0, X), para todos os símbolos $X \in (V \cup \Sigma)$, e descártanse os conxuntos baleiros. Obtéñense I_1, I_2, ...
4) Aplicámoslle Ir_a de novo a I_1, I_2, ... ata non obter novos conxuntos.

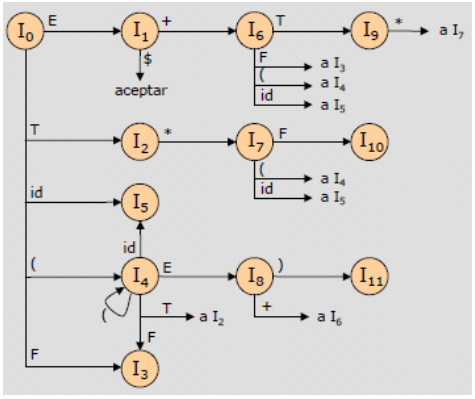
Exemplo:

I_6=Ir_a(I_1,+)	I_7=Ir_a(I_2,*)	I_8=Ir_a(I_4,E)	I_9=Ir_a(I_6,T)	I_{10}=Ir_a(I_7,F)	I_{11}=Ir_a(I_8,))
E → E+·T					
T → ·T*F	T → T*·F	F → (E·)	E → E+T·	T → T*F·	F → (E)·
T → ·F	F → ·(E)	E → E·+T	T → T·*F		
F → ·(E)	F → ·id				
F → ·id					

Se consideramos cada conxunto I_j como un estado dun autómata finito, Ir_a proporciónanos a súa táboa de transicións:

Exemplo:

Estado	id	+	*	(	)	$	E	T	F
0	5			4			1	2	3
1		6							
2			7						
3									
4	5			4			8	2	3
5									
6	5			4				9	3
7	5			4					10
8		6			11				
9			7						
10									
11									



Pazo

ANÁLISE SINTÁCTICA ASCENDENTE (ASA)

mércores, 6 de marzo de 2024 09:07

táboa de accións SLR(1)

Volvemos á pregunta clave: *cómo sabe un analizador sintáctico ascendente cándo desprazar e cándo reducir?*

Podemos construír unha **táboa de accións** que nos permitirá tomar esta decisión. O procedemento é o seguinte:

1) Construír a colección canónica LR(0) = { I_0, I_1, ..., I_n }.
2) Para os elementos de I_j da forma A→ $\alpha \cdot a\beta$, onde a é un terminal, e Ir_a(I_j, a) = I_k, entón acción[j, a] = **desprazar** a entrada e ir ao estado correspondente a I_k.
3) Para os elementos de I_j da forma A→ $\alpha \cdot$, entón acción[j, s] = **reducir** A→ α, para calquera
4) s ∈ SEGUINTES(A).
5) Se o elemento $S' \rightarrow S \cdot \$$ está en I_j, entón acción[j, $] = **aceptar**.
6) As entradas baleiras da táboa de accións representan estados de **erro**.

Exemplo:

E′ → E$
(1) E → E + T
(2) E → T
(3) T → T * F
(4) T → F
(5) F → (E)
(6) F → id

	Acciones						Ir_a		
Estado	id	+	*	(	)	$	E	T	F
0	d5			d4			1	2	3
1		d6				OK			
2		r2	d7		r2	r2			
3		r4	r4		r4	r4			
4	d5			d4			8	2	3
5		r6	r6		r6	r6			
6	d5			d4				9	3
7	d5			d4					10
8		d6			d11				
9		r1	d7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

ANÁLISE SINTÁCTICA ASCENDENTE (ASA)

procedemento de análise

Na táboa anterior, **dn** significa desprazar a entrada e ir ao estado n e **rn** significa reducir mediante a regra (n).

Para entender mellor os desprazamentos e as reducións observamos un exemplo da táboa:

$$I_9 = Ir\_a(I_6, T) = \{E \rightarrow E + T\cdot\ ,\ T \rightarrow T \cdot * F\}$$

Significa que, no estado 6, se analizamos T debemos ir ao estado 9, caracterizado pola aplicación de dúas regras distintas:

1. No caso de $E \rightarrow E + T \cdot$, a acción será **reducir** pola regra (1) cos símbolos de SEGUINTES(E) = { +,), $ }.
2. No caso de $T \rightarrow T \cdot * F$, a acción será **desprazar** a entrada e ir ao estado 7, xa que $I_7 = Ir\_a(I_9, *)$.

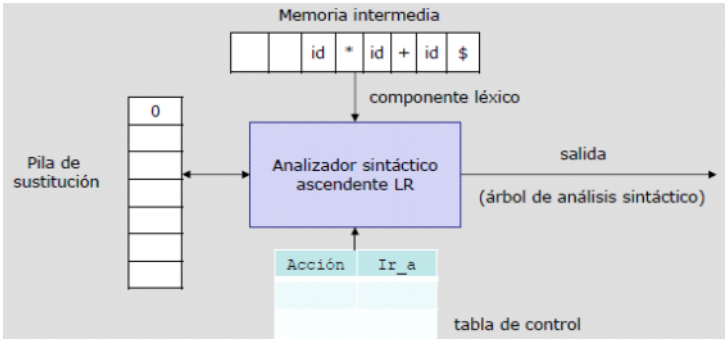
Pazo

ANÁLISE SINTÁCTICA ASCENDENTE (ASA)

mércores, 6 de marzo de 2024 09:28

procedemento de análise

Podemos proporcionar un procedemento de análise baseado no uso dunha táboa de control para seleccionar a seguinte regra de substitución e unha pila para almacenar os símbolos gramaticais.



O algoritmo de análise é o seguinte:

1) Inicializar a pila co estado 0.
2) Para cada estado da pila i e símbolo da entrada a consultamos a cela acción[i, a].
   i) Se acción[i, a] = dn, entón **desprazamos** a entrada, imos ao estado n e introducímolo no cumio da pila.
   ii) Se acción[i, a] = rn, entón **reducimos** mediante a regra (n), da forma $A \rightarrow \beta$; para o que, se m é a lonxitude de β, eliminamos m estados do cumio da pila e, a continuación, imos ao estado Ir_a(h, A), onde h é o estado que queda na pila ao eliminar os m estados superiores.
   iii) Se acción[i, a] = **aceptar**, completamos a análise.
   iv) Se acción[i, a] = Ø, **erro**, e invocamos a xestión de erros.

ANÁLISE SINTÁCTICA ASCENDENTE (ASA)

conflitos nas táboas de accións SLR(1)

Unha gramática é **SLR(1)** se na táboa de análise **aparece como máximo unha entrada en cada cela**. Cando non é así, pódense sinalar dous tipos de conflitos:

1. **Desprazamento-redución**. Sucede cando na mesma cela aparece unha acción de desprazar e unha de reducir. Normalmente, para poder continuar a análise, escóllese unha acción por defecto -xeralmente a de desprazar-.
2. **Redución-redución**. Ocorre cando, nun estado da táboa, a mesma entrada se pode reducir por dúas regras distintas. A solución máis razoable é modificar a gramática para que isto non ocorra.

*Exemplo: desenvolver a análise de id*id+id$.*

Estados	Símbolos	Entrada	Acción
0		id*id+id$	Desplazar e ir al estado 5
0 5	id	*id+id$	Reducir con (6) F → id (se quita 5 de la pila y se va a Ir_a(0,F)=3)
0 3	F	*id+id$	Reducir con (4) T → F (se quita 3 de la pila y se va a Ir_a(0,T)=2)
0 2	T	*id+id$	Desplazar e ir al estado 7
0 2 7	T*	id+id$	Desplazar e ir al estado 5
0 2 7 5	T*id	+id$	Reducir con (6) F → id (se quita 5 y se va a Ir_a(7,F)=10)
0 2 7 10	T*F	+id$	Reducir con (3) T → T * F (se quitan 10, 7, 2 y se va a Ir_a(0,T)=2
0 2	T	+id$	Reducir con (2) E → T (se quita 2 y se va a Ir_a(0,E)=1)
0 1	E	+id$	Desplazar e ir al estado 6
0 1 6	E+	id$	Desplazar e ir al estado 5
0 1 6 5	E+id	$	Reducir con (6) F → id (se quita 5 y se va a Ir_a(6,F)=3)
0 1 6 3	E+F	$	Reducir con (4) T → F (se quita 3 y se va a Ir_a(6,T)=9)
0 1 6 9	E+T	$	Reducir con (1) E → E + T (se quitan 9, 6, 1 y se va a Ir_a(0,E)=1)
0 1	E	$	Aceptar

Pazo

ANÁLISE SINTÁCTICA ASCENDENTE (ASA)

mércores, 6 de marzo de 2024 09:55

xestión de erros en analizadores SLR(1)

Os erros sintácticos prodúcense cando, a partir do estado actual situado no cumio da pila, non hai ningunha transición definida para o símbolo actual da entrada. Nesta situación, o estado da pila representa o **contexto á esquerda** do erro, e o resto da cadea de entrada, o **contexto á dereita**.

A recuperación do erro lévase a cabo modificando a pila e/ou a cadea de entrada, ata que o estado e a cadea lle permiten ao analizador continuar a análise.

Mostraremos dous **métodos** de xestión de erros:

- Métodos **heurísticos**.
- Método de **análise de transicións**.

→ Os **métodos heurísticos** supoñen a programación de rutinas específicas para cada cela na que se produce un erro.

→ O método de **análise de transicións** propón explorar a pila cara abaixo ata atopar un estado d cun Ir_a para un non terminal A. Despois, descártanse cero ou máis símbolos na entrada ata atopar un que poida seguir a A de acordo coa gramática. Iso permite meter o estado Ir_a(d, A) na pila e continuar coa análise.

Pazo

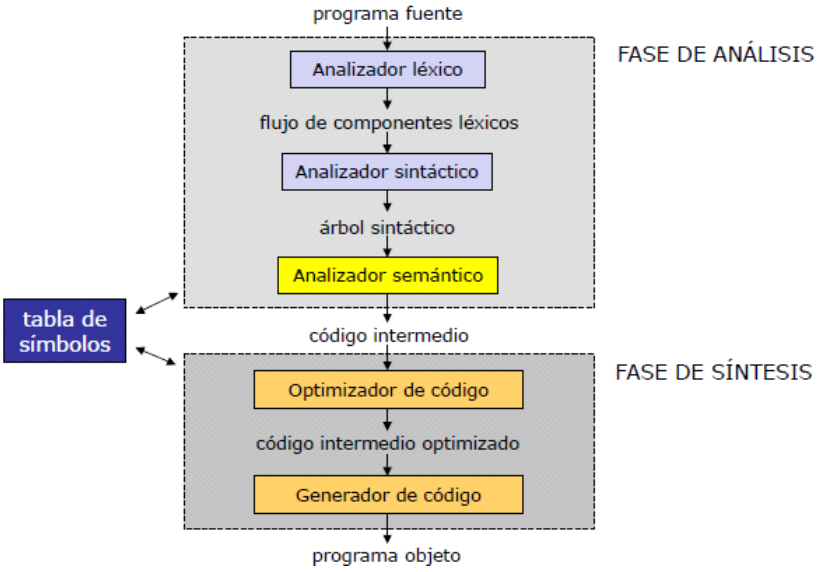
INTRODUCIÓN Á ANÁLISE SEMÁNTICA

luns, 18 de marzo de 2024 18:50

Podemos entender a **semántica** dunha linguaxe de programación coma o conxunto de regras que especifican o significado de calquera sentencia sintacticamente correcta.

- A análise semántica pódese levar a cabo como unha tarefa de análise independente das demais, ou ben en paralelo á análise sintáctica.
- Como **saída** da análise semántica xérase automaticamente unha representación en forma de **código intermedio** (independente da máquina de execución) que pasa ás fases de síntese da compilación.

ESTRUTURA



PRELIMINARES

- Cada símbolo da gramática adquire un determinado significado segundo o contexto de aparición no código fonte.
- Para que os símbolos poidan ter significado, é necesario aumentar a gramática e engadirlle un conxunto de **atributos** aos símbolos e unhas **accións** semánticas ás regras sintácticas.
- A especificación da semántica pódese realizar mediante:
  - **Linguaxes especiais de especificación**, coma VD, (para PL/1), Anna (para Ada), Gypsy ou HDM.
  - **Gramáticas de atributos**, que serán as que describiremos neste tema. Chamarémoslle atributo dun símbolo da gramática a todo ítem de información engadido á árbore de derivación durante a análise semántica.

Pazo

GRAMÁTICAS DE ATRIBUTOS

luns, 18 de marzo de 2024 20:22

Chamámoslle **gramática de atributos** a toda aquela gramática independente do contexto á que se lle engade un sistema de atributos.

Un sistema de atributos está formado por:

1. Un conxunto de **atributos semánticos** asociados a cada símbolo da gramática.
2. Un conxunto de **accións semánticas**, distribuídas ao longo das regras de substitución.

→ Un **atributo** dun símbolo X é unha **variable** que representa unha determinada **característica** do símbolo e que pode tomar un valor de entre un conxunto, finito ou non, de valores posibles. Denotaremos o atributo "a" do símbolo X como "X.a".

→ Unha **acción semántica** é un **algoritmo** asociado a unha **regra** da gramática, que ten o obxectivo de calcular o valor de algún dos atributos dos símbolos de dita regra.

Por exemplo:

$D \rightarrow TL;$
$T \rightarrow int \mid real$
$L \rightarrow L, id \mid id$

Unha posible gramática de atributos sería:

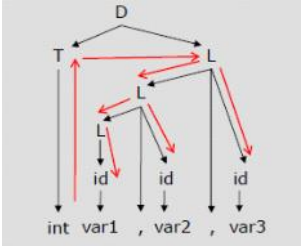
```
D → TL;
    { L.tipo=T.tipo; }
T → int
    { T.tipo=entero; }
T → real
    { T.tipo=real; }
L → Ld,id
    { Ld.tipo=L.tipo; insertarTS(id,L.tipo); }
L → id
    { insertarTS(id,L.tipo); }
```

PROPAGACIÓN DE ATRIBUTOS

Chamámoslle **propagación** ao cálculo do valor dun atributo en función do valor doutros. Defínese así unha relación de dependencia entre os atributos que aparecen na árbore de análise.

Exemplo:

Propagar o atributo "tipo" na declaración indicada segundo a gramática anterior:



TRADUCIÓN DIRIXIDA POLA SINTAXE

A propagación realízase seguindo a árbore de derivación que devolve a análise sintáctica, e incluso se pode desenvolver ao mesmo tempo.

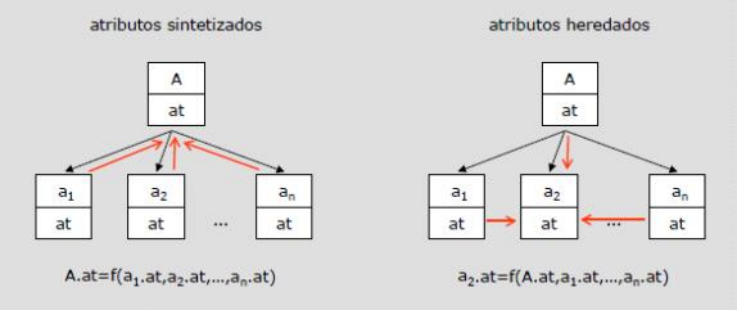
Segundo os símbolos que interveñen nunha acción semántica, para calcular o valor do atributo dun símbolo, falamos de dous tipos de atributos:

1. **Atributos sintetizados**. Son atributos asociados a non terminais da parte esquerda dunha regra de substitución. Calcúlanse a partir dos nodos fillos da subárbore na que aparecen. A información percorre a árbore verticalmente.
2. **Atributos herdados**. Son atributos asociados a símbolos da parte dereita dunha substitución. Calcúlanse a partir dos atributos do nodo pai e dos nodos irmáns dentro da mesma subárbore. A información percorre a árbore horizontalmente.

Pazo

TRADUCIÓN DIRIXIDA POLA SINTAXE

luns, 18 de marzo de 2024 20:39



GRAFO DE DEPENDENCIAS

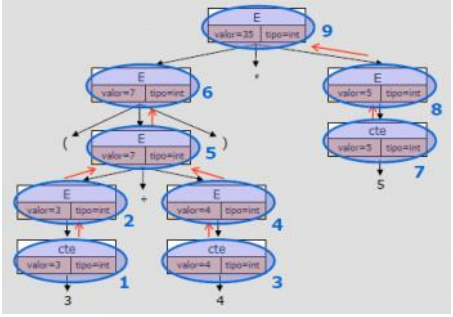
A propagación débese realizar seguindo a orde implícita na dependencia que existe entre os atributos. Non podemos avaliar unha regra semántica antes de dispor de todos os atributos involucrados.

Esta orde represéntase mediante un **grafo de dependencias acíclico e dirixido**, que construímos representando as dependencias entre atributos da forma $b = f(c_1, c_2, \dots, c_n)$.

O grafo terá un nodo para cada atributo de cada regra semántica, e un arco de c_i a b se b depende de c_i. Así:

1) Avalíanse os nodos aos que non chega ningún arco e márcanse como calculados.
2) Avalíanse os nodos nos que todos os arcos proceden de nodos marcados, e márcanse como calculados.
3) Continuamos ata que non quedan máis nodos por avaliar.

Unha vez construído o grafo de dependencias débese determinar unha **orde** compatible coas dependencias.



VERIFICACIÓN DE TIPOS

- A **verificación de tipos** asegura que o tipo de cada unha das construcións do código fonte coincide co previsto no deseño da gramática.

- Un **sistema de tipos** é unha serie de regras que permiten asignarlle tipos ás distintas construcións do código fonte: en primeiro lugar os **tipos básicos** (enteiro, real, carácter, etc.), e a continuación, **expresións de tipos** para matrices, estruturas, punteiros, funcións, etc..

Pazo

VERIFICACIÓN DE TIPOS

luns, 18 de marzo de 2024 22:25

- Estas regras inclúense na árbore de análise sintáctica, dun modo similar ás de asignación e propagación de atributos.

- Segundo a verificación de tipos, distínguese entre linguaxes forte e debilmente tipificados, ou linguaxes con verificación estática e dinámica de tipos:
  - As linguaxes **fortemente tipificadas**, serían aqueles nos que non é posible un erro de tipo en tempo de execución. *Por exemplo: Lisp, Java, Perl, Python, ...*
  - As linguaxes **debilmente tipificadas**, serían aqueles nos que é posible empregar un valor dun tipo coma se fose doutro tipo. *Por exemplo: C.*
  - As linguaxes con **tipificación estática** son aqueles que realizan a verificación de tipos sobre o código fonte, é dicir, en tempo de compilación. *Por exemplo: C++, D, Go, Kotlin, Java, ...*
  - As linguaxes con **tipificación dinámica** son aquelas que só realizan a verificación de tipos en tempo de execución. *Por exemplo: Pyhton, Javascript, Julia, R, MATLAB, ...*

Exemplo:

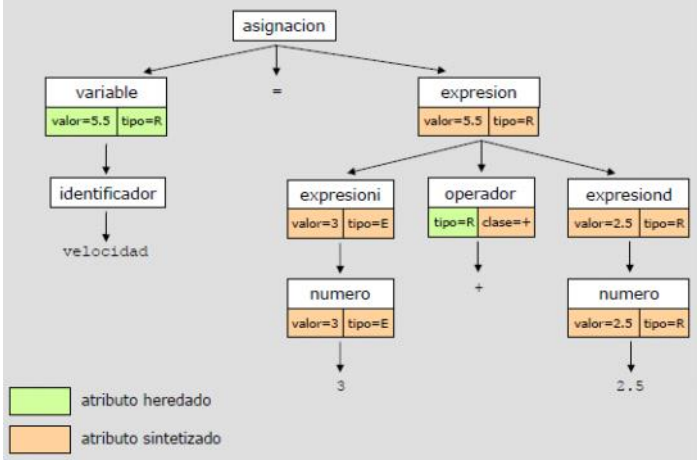
Sexa a seguinte gramática:

```
<asignacion> ::= <variable> = <expresion>
 <expresion> ::= <expresion> <operador> <expresion> | numero
  <variable> ::= identificador
  <operador> ::= + | -
```

Construiremos unha gramática de atributos para realizar sumas e restas de números reais e enteiros. Mostraremos a árbore sintáctica para a entrada `velocidad = 3+2.5`*, e aplicaremos as accións semánticas para obter o resultado.*

```
<asignacion> ::= <variable> = <expresion>
     { variable.valor = expresion.valor;
       variable.tipo = expresion.tipo; }
 <expresion> ::= <expresioni> <operador> <expresiond>
     { operador.tipo = MayorTipo(expresioni.tipo,expresiond.tipo);
       expresion.tipo = operador.tipo;
       if (operador.tipo = 'real' AND expresioni.tipo = 'entero')
          {expresioni.valor = float(expresioni.valor); }
       if (operador.tipo = 'real' AND expresiond.tipo = 'entero')
          {expresiond.valor = float(expresiond.valor); }
       switch (operador.tipo) {
         'entero': expresion.valor=OpEntera(operador.clase,expresioni.valor,expresiond.valor);
                   break;
          'real': expresion.valor=OpReal(operador.clase,expresioni.valor,expresiond.valor);
                   break; }
     }
 <expresion> ::= numero
     { expresion.valor = numero.valor;
       expresion.tipo = numero.tipo; }
  <variable> ::= identificador
  <operador> ::= + { operador.clase = '+'; }
  <operador> ::= - { operador.clase = '-'; }
```

```
char MayorTipo (char tipo1, char tipo2)
{ char tipo;
  if ((tipo1 == 'real') || (tipo2 == 'real'))
      tipo = 'real';
  else
      tipo = 'entero';
  return tipo;
}
int OpEntera (char op, int valor1, int valor2)
{ int resultado;
  switch (op) {
     '+': resultado = (valor1 + valor2); break;
     '-': resultado = (valor1 - valor2); break;  }
  return resultado;
}
float OpReal (char op, float valor1, float valor2)
{ float resultado;
  switch (op) {
     '+': resultado = (valor1 + valor2); break;
     '-': resultado = (valor1 - valor2); break;  }
  return resultado;
}
```



Pazo

XERACIÓN DE CÓDIGO INTERMEDIO

luns, 18 de marzo de 2024 23:05

Unha representación intermedia non é máis ca unha estrutura de datos que representa ao código fonte durante o proceso de tradución a código obxecto.

Ata agora, empregamos a **árbore de análise sintáctica** xunto coa táboa de símbolos como representación do código fonte. Aínda que é unha representación válida, parécese pouco a un código executable.

A **notación postfixa** tamén é unha representación intermedia empregada frecuentemente na tradución de expresións. Propomos unha representación intermedia como resultado dunha linealización da árbore de análise sintáctica, chamada **código de tres direccións**.

Existen moitos de códigos de tres direccións. Todos eles desenvólvense para algún tipo de **máquina virtual**.

XERACIÓN DE CÓDIGO INTERMEDIO

código de tres direccións

O código de tres direccións defínese como unha **secuencia de instrucións** da forma: `x = y op z`

onde `op` representa a calquera operador (aritmético, lóxico, etc.) e `x`, `y` e `z` representan símbolos definidos polo programador (residentes na táboa de símbolos), variables temporais xeradas polo compilador, constantes, etc..

Esta representación esíxenos **descompor** e **modificar** as expresións complexas e as sentencias de fluxo de control para dar lugar a instrucións de tres direccións. O código de tres direccións é unha **representación linealizada de esquerda a dereita da árbore de análise sintáctica**, onde lle asignaremos nomes temporais aos nodos internos da árbore.

Por exemplo:

Mostramos dous exemplos de tradución a código de tres direccións de dúas expresións:

(a + b) / (c + e)	z := x + y + z / y
t1 := a + b	t1 := x + y
t2 := c + e	t2 := z / y
t3 := t1 / t2	t3 := t1 + t2
	z := t3

- As instrucións de tres direccións xéranse mediante regras semánticas e o uso dun atributo .código.
- O proceso de tradución do código de tres direccións a código máquina adoita ser bastante sinxelo.

Pazo

XERACIÓN DE CÓDIGO INTERMEDIO

martes, 19 de marzo de 2024 20:11

código de tres direccións

Para poder representar as construcións dunha linguaxe de alto nivel, é necesario introducir no código intermedio operadores de representación de saltos condicionais, incondicionais, chamadas a funcións, punteiros, etc.:

- **Instrucións de asignación** con operador **binario**, onde op é unha operación aritmética ou lóxica:
  ```
  x := y op z
  ```
- **Instrucións de asignación** con operador **unitario**, onde op é unha operación de oposto, negación, desprazamento ou conversión de tipo:
  ```
  x := op z
  ```
- **Instrucións de copia**:
  ```
  x = y
  ```
- **O salto incondicional**, onde se emprega unha etiqueta E para indicar a seguinte instrución a executar:
  ```
  goto E
  ```
- **O salto condicional**, onde se executa a instrución de etiqueta E ou a seguinte na secuencia de instrucións dependendo do resultado dun operador de relación:
  ```
  if x oprel y goto E
  ```
- **Chamadas a procedementos**, da forma $p(x_1, x_2, \dots, x_n)$:
  ```
  param x1
  ...
  param xn
  call p, n
  ```
  e devolucións, mediante unha instrución `return y`.
- **Asignacións con índices**, de dúas formas distintas:
  ```
  x := y[i]
  x[i] := y
  ```
- **Asignacións de direccións e punteiros**:
  ```
  x := &y
  x := *y
  *x := y
  ```

Un conxunto de operadores pequeno é máis doado de implementar nunha máquina obxecto. Non obstante, se resulta ser un conxunto limitado pode obrigar a xerar longas secuencias de instrucións para algunhas construcións da linguaxe fonte.

Exemplo:

Mostramos un exemplo de tradución a código de tres direccións:

lenguaje de alto nivel	código de tres direcciones
`read x;`	`100 read x`
`if x > 0 then`	`101 iffalse x > 0 goto L1`
`  fact := 1;`	`102 fact = 1`
`  repeat`	`103 label L2`
`  fact := fact * x;`	`104 t2 = fact * x`
`  x := x - 1;`	`105 fact = t2`
`  until x = 0;`	`106 t3 = x - 1`
`  write fact;`	`107 x = t3`
`end;`	`108 iffalse x = 0 goto L2`
	`109 write fact`
	`110 label L1`
	`111 halt`

Pazo

XERACIÓN DE CÓDIGO INTERMEDIO

martes, 19 de marzo de 2024 20:26

implementacións

O código intermedio adoita almacenarse como unha **lista enlazada de rexistros**, onde cada rexistro representa unha instrución. Existen dúas implementacións diferentes: con cuádruples ou con triples.

- Un **cuádruple** é unha estrutura de tipo rexistro con catro campos, que chamaremos `op`, `result`, `arg1` e arg2 . Por exemplo, a instrución `x = y + z` represéntase como (`suma`, `x`, `y`, `z`). As instrucións que non empregan todos os campos deixan baleiros os que non empregan.

- Un **triple** é unha simplificación na que cada instrución representa a unha variable temporal, empregando así só tres campos: `op`, `arg1`, `arg2`. Por exemplo, a instrución `x = y + z` represéntase como `01(suma, y, z)` e `02(asigna, x, (01))`.

Exemplo:

Mostramos un exemplo de tradución mediante cuádruples e triples.

lenguaje de alto nivel	código de tres direcciones	cuádruples	triples
read x;	100 read x	(read,x,,)	(100) (read,x,)
if x > 0 then begin	101 iffalse x > 0 goto L1	(ble,x,0,L1)	(101) (gt,x,0)
fact := 1;	102 fact = 1	(assign,fact,1,)	(102) (iffalse,(101),(111))
repeat	103 label L2	(label,L2,,)	(103) (assign,fact,1)
fact := fact * x;	104 t2 = fact * x	(mult,t2,fact,x)	(104) (mult,fact,x)
x := x - 1;	105 fact = t2	(assign,fact,t2,)	(105) (assign,fact,(104))
until x = 0;	106 t3 = x - 1	(sub,t3,x,1)	(106) (sub,x,1)
write fact;	107 x = t3	(assign,x,t3,)	(107) (assign,x,(106))
end;	108 iffalse x = 0 goto L2	(bne,x,0,L2)	(108) (eq,x,0)
	109 write fact	(write,fact,,)	(109) (iffalse,(108)(104))
	110 label L1	(label,L1,,)	(110) (write,fact,)
	111 halt	(halt,,,)	(111) (halt,,)

O uso de cuádruples ten as súas vantaxes respecto aos triples cando a optimización de código supón a reorganización de código: **se modificamos as posicións dos triples temos que modificar as súas referencias.**

Para evitar este problema introdúcense os **triples indirectos**: unha lista de punteiros a triples que evita modificar as posicións destes últimos. Calquera optimización de código realízase sobre esta lista.

Lista de punteros	punteros	triples
(1)	(100)	(100) (read,x,)
(2)	(101)	(101) (gt,x,0)
(3)	(102)	(102) (iffalse,(101),(111))
(4)	(103)	(103) (assign,fact,1)
(5)	(104)	(104) (mult,fact,x)
(6)	(105)	(105) (assign,fact,(104))

Pazo

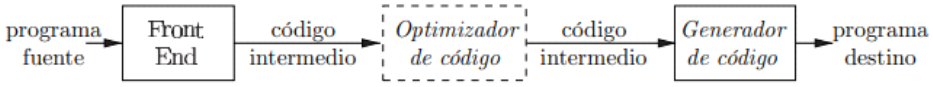
INTRODUCIÓN Á XERACIÓN DE CÓDIGO

mércores, 20 de marzo de 2024 12:09

! A información deste tema sae das diapositivas de clase e mais do capítulo 8 do Aho

Entrada:

- Representación intermedia
- Táboa de símbolos



Saída:

- Programa destino semánticamente correcto e "óptimo"

Fase adicional de optimización:

- Problemas NP-completos
- Uso de heurísticas

TAREFAS PRINCIPAIS

- Selección de instrucións
- Repartición e asignación de rexistros
- Ordenamento de instrucións

O criterio principal é a xeración de **código correcto**.
O secundario, xerar **código de calidade**.

Entrada do Xerador de Código

A entrada do xerador de código é a **representación intermedia** de nivel baixo do programa fonte, xunto coa información da táboa de símbolos. Suporemos que se detectaron e corrixiron todos os **erros** sintácticos e semánticos estáticos, que se levou a cabo a **comprobación de tipos** de datos necesaria e que se insertaron **operadores de conversión** de tipos en todos os lugares necesarios.

O Programa Destino

A **arquitectura** do conxunto de instrucións da máquina destino ten un impacto considerable na dificultade de construír un bo xerador de código que produza código máquina de alta calidade. As arquitecturas máis comúns son **RISC** (Reduced Instruction Set Computer), **CISC** (Complex Instruction Set Computer) e baseadas en **pilas**.

Selección de Instrucións

O xerador de código debe asignarlle o programa de representación intermedia a unha secuencia de código que a máquina de destino poida executar. A complexidade de realizar esta asignación depende do **nivel** da representación intermedia, da natureza da **arquitectura** do conxunto de instrucións e da **calidade** desexada.

Se a representación intermedia é de alto nivel, o xerador de código pode traducir cada instrución deste tipo nunha secuencia de instrucións de máquina empregando **plantillas de código**.

p.e.:

x=y+z	LD R0, y	// Carga y en el registro R0
	ADD R0, R0, z	// Suma z a R0
	ST x, R0	// Almacena R0 en x

Pazo

Asignación de Rexistros

martes, 7 de maio de 2024 23:32

Un problema clave na xeración de código e decidir que valores gardar en que **rexistros**. Os rexistros son a unidade computacional máis veloz na máquina destino, pero, en xeral, non chegan para conter todos os valores. Aqueles que non se almacenen en rexistros deberán ir á memoria.

Frecuentemente, o uso dos rexistros divídese en dous subproblemas:

- **Repartición de rexistros**, durante a que seleccionamos o conxunto de variables que residirán nos rexistros en cada punto do programa.
- **Asignación de rexistros**, durante o que escollemos o rexistro específico no que residirá unha variable.

É complicado atopar unha asignación óptima de rexistros a variables. En sentido matemático, o problema é **NP-completo**.

Ademais, inflúen outros factores coma o **hardware** da máquina destino. Por exemplo, pares de rexistros de propósito específico para multiplicar ou dividir.

Orde de Avaliación

A **orde** na que se realizan os cálculos pódelle afectar á eficiencia do código destino, xa que podería **reducir o número de rexistros necesarios**, permitir **explotar paralelismo** a nivel de instrucións, fíos ou núcleos e mais **mellorar o rendemento** en termos xerais. Non obstante, elixir a mellor orde adoita ser un problema **NP-completo difícil**.

A LINGUAXE DESTINO: un modelo simple de máquina destino

A nosa computadora destino terá as seguintes características:

- n rexistros (r): R0, R1, ..., Rn-1
- Direcionamento por bytes
- Operandos enteiros
- Tipos de instrucións
  - LD r, x
  - LD r1, r2
  - ST x, r
  - ADD r1, r2, r3
  - BR L
  - Bcond r, L
- Modos de direcionamento:
  - Nome de dirección en memoria dunha variable x
  - Indexada a(r) é contido de (a + contido(r))
  - Indexada 100(r)
  - Indirecto *r
  - Indirecto *100(r) é contido de (contido(100+contido(r)))
  - Inmediato #100
- Comentarios //

x=*p

```
LD R1, p          // R1=p
LD R2, 0(R1)      // R2=cont(0+cont(R1))
ST x, R2          // x=R2
```

If (x<y) goto L

```
LD R1, x          // R1=x
LD R2, y          // R2=y
SUB R1, R1, R2    // R1=R1-R2
BLTZ R1, L        // if R1<0 salta a L
```

x=y-z

```
LD R1, y          // R1=y
LD R2, z          // R2=z
SUB R1, R1, R2    // R1=R1-R2
ST x, R1          // x=R1
```

Pazo

Melloras da Calidade do Código

mércores, 8 de maio de 2024 11:46

Dependendo do aspecto do programa que nos interese optimizar, algunhas medidas de **custos** comúns das instrucións e do programa son:

- **Tempo** de **compilación**
- **Tamaño** do código compilado
- **Tempo** de **execución**
- **Consumo** de enerxía

A busca dun programa destino **óptimo** é un problema indecidible, e moitos dos problemas involucrados son **NP-completos**. Na xeración de código debemos, frecuentemente, contentarnos coas técnicas de **heurística** que producen bos programas destino, pero que non necesariamente son óptimos.

No **Aho**, defínese o **custo dunha instrución** coma o número de palabras da mesma. Unha definición **alternativa** sería o número de accesos a memoria que conleva.

Direccións no Código Destino

A conversión de nomes da representación intermedia en direccións lóxicas (virtuais) no código destino. O espazo de direccións lóxicas de cada programa particiónase en catro áreas de código e catro de datos:

- Unha área de **Código** determinada en forma estática, que contén o código destino executable. O tamaño deste código pódese determinar en tempo de compilación.
- Unha área de datos **Estática** determinada en forma estática, para conter variables globais e demais datos que xera o compilador. O tamaño destes datos pode determinarse en tempo de compilación.
- Unha área **Montículo** (**Heap**) administrada en forma dinámica para conter obxectos de datos que se asignan e se liberan durante a execución do programa. O seu tamaño non se pode determinar en tempo de compilación.
- Unha área **Pila** (**Stack**) administrada en forma dinámica, para conter os rexistros de activación a medida que se crean e se destrúen, durante as chamadas aos procedementos e os seus retornos. O seu tamaño non se pode determinar en tempo de compilación.

A asignación de chamadas a rutinas pode ser estática ou converterse en asignación de pila mediante o uso de direccións relativas para o almacenamento nos rexistros de activación. (+ info: 8.3.1-2 Aho)

BLOQUES BÁSICOS E GRAFOS DE FLUXO

Os **grafos de fluxo** son unha representación gráfica do código intermedio. Os seus nodos son **bloques básicos** e as súas frechas indican a **orde** de execución dos bloques.

Estes grafos son útiles para a caracterización do código e a aplicación de técnicas de mellora de calidade en bloques e entre bloques.

Bloque básico:

- O fluxo de control entra a través da primeira instrución do bloque, non hai saltos á parte media do bloque.
- O control sae do bloque sen deterse* nin bifurcarse agás na última instrución do bloque.

* ***Que pasa coas interrupcións?*** *Resposta curta: non debemos preocuparnos. Ou o control volve ao bloque ou se produce un erro. En calquera caso, non entra dentro das competencias da optimización.*

Algoritmo para Obter Bloques Básicos

mércores, 8 de maio de 2024 13:24

```
1)  i = 1
2)  j = 1
3)  t1 = 10 * i
4)  t2 = t1 + j
5)  t3 = 8 * t2
6)  t4 = t3 - 88
7)  a[t4] = 0.0
8)  j = j + 1
9)  if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```

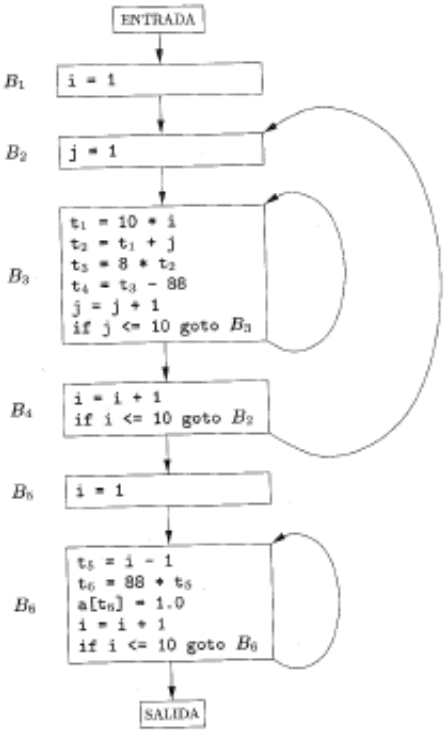


Figura 8.9: Grafo de flujo de la figura 8.7

- Hai un nodo por cada bloque básico
- Unha frecha se hai un posible salto do final dun bloque ao comezo doutro.
- Unha frecha se van seguidos e o primeiro non remata nun salto incondicional.
- Engádense nodos de entrada e saída.

- Nos bloques básicos pódense realizar cambios considerables.
- Entre bloques básicos pódense aplicar técnicas de optimización.

Ciclos

É importante identificar os ciclos do código, xa que consumen a maior parte do tempo de execución. A partir do diagrama de fluxo, é doado identificalos:

- Un conxunto de nodos é un ciclo se hai un nodo de entrada único.
- Cada nodo ten unha ruta non baleira e completamente dentro do ciclo que vai á entrada.

Pazo

Información de seguinte uso

mércores, 8 de maio de 2024 13:56

É esencial saber cándo se vai usar o valor dunha variable para xerar bo código. **Unha variable está viva nunha instrución se é empregada posteriormente por outra.** Se o valor dunha variable que se atopa nun rexistro nunca se empregará máis adiante, entón ese rexistro pode asignárselle a outra variable.

O Algoritmo 8.7 serve para determinar a información sobre a vida e o uso seguinte para cada instrución nun bloque básico.

1. z=x+b
2. y=a*c
3. x=y+z

	a	b	c	x	y	z
	v	v	v	v	v	v
	0	0	0	0	0	0
3.	v	v	v	**nv**	**v**	**v**
	0	0	0	**0**	**3**	**3**
2.	**v**	v	**v**	nv	**nv**	v
	2	0	**2**	0	**0**	3
1.	v	**v**	v	**v**	nv	**nv**
	2	**1**	2	**1**	0	**0**

Optimización dos Bloques Básicos

Representación DAG

Moitas técnicas para a optimización local comezan coa **representación en DAG (grafo dirixido acíclico) dos bloques básicos**:

- Un nodo para cada valor inicial das variables.
- Un nodo para cada instrución cuxos fillos son as máis recentes definicións dos operandos.
- Etiquétase cada nodo de instrución co operador e adxúntase a lista de variables actualizadas por ela.
- Certos nodos caracterízanse como nodos saída (as súas variables están vivas ao saír do bloque).

Busca de subexpresións locais comúns

Aquelas expresións que calculan un valor previamente calculado:

a=b+c
b=a-d
c=b+c
d=a-d

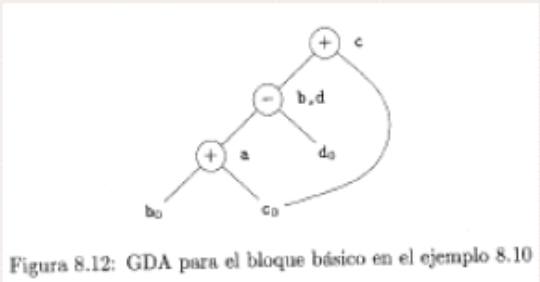


Figura 8.12: GDA para el bloque básico en el ejemplo 8.10

a=b+c
d=a-d
c=d+c

a=b+c
b=b-d
c=c+d
e=b+c

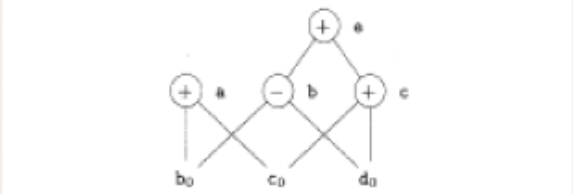


Figura 8.13: GDA para el bloque básico en el ejemplo 8.11

Pazo

Eliminación de código morto

mércores, 8 de maio de 2024 14:08

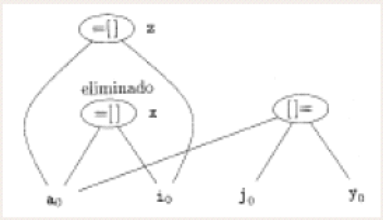
Suprímense aqueles nodos que non teñen variables vivas adxuntas.

Uso de indentidades alxebraicas

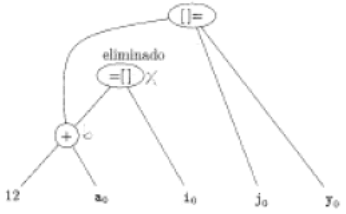
- Identidades aritméticas
- Redución por forza local
- Pregado de constantes
- Conmutatividade e asociatividade (con coidado!)

Representación de referencias a arrays

x=a[i]
a[j]=y
z=a[i]



b=12+a
x=b[i]
b[j]=y



Asignacións de punteiros e chamadas a procedementos

- O operador =* (x = *p) debe aceptar todos os nodos que se atopan asociados como argumentos.
- O operador *= (*q = y) elimina todos os nodos construídos ata agora no DAG.
- Casos particulares: p=&x *p=y

Reensamblado dos bloques básicos a partir do DAG

- 5 regras
- Se o nodo ten máis dunha variable viva adxunta, debemos introducir instrucións de copia para darlle o valor correcto a cada unha desas variables.

a=b+c
b=a-d
c=b+c
d=a-d

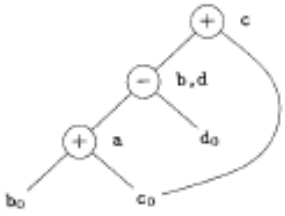


Figura 8.12: GDA para el bloque básico en el ejemplo 8.10

Se b non está viva:

a=b+c
d=a-d
c=d+c

Se b está viva:

a=b+c
d=a-d
b=d
c=b+c

Pazo

UN XERADOR DE CÓDIGO SINXELO

mércores, 8 de maio de 2024 15:42

Imos considerar un algoritmo que xera código para un só bloque básico.
As instrucións de máquina son da seguinte forma:

- LD reg, mem (carga)
- ST mem, reg (almacenamento)
- OP reg, reg, reg (operación)

Supomos tamén que para cada operador hai exactamente unha instrución de máquina que recibe os operandos necesarios nos rexistros e que realiza a operación correspondente, deixando o resultado nun rexistro.

Descritores de Rexistros e Direccións

- Para cada rexistro almacénanse os nomes das variables das que o valor está almacenado en dito rexistro (descritor de rexistro).
- Para cada variable almacénanse as posicións onde se pode atopar (na TS) (descritor de acceso).

Algoritmo de Xeración de Código

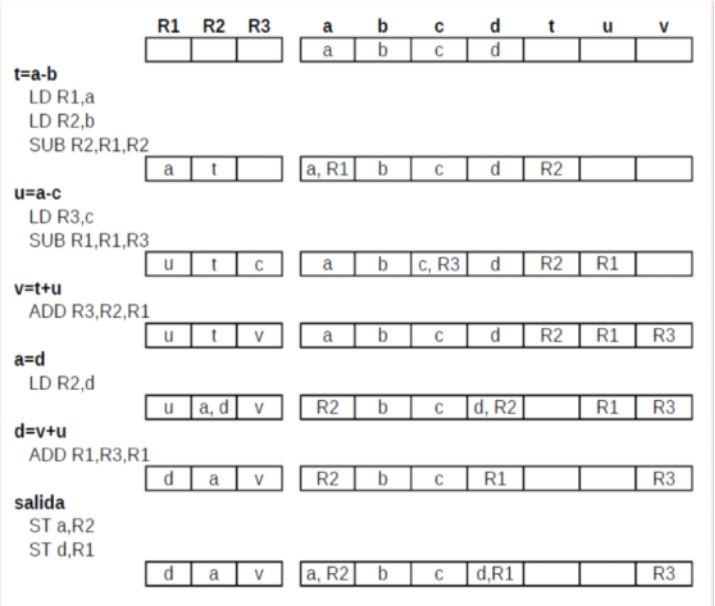
- Emprega unha función denominada "obtenReg" para seleccionar os rexistros para cada instrución da representación de tres direccións.
- Ao final do bloque básico, xera almacenamentos para as variables vivas.
- Regras de administración dos descritores de rexistros e de direccións.

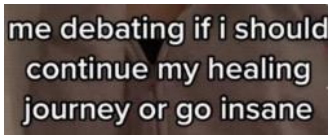
p.e.:

```
t=a-b
u=a-c
v=t+u
a=d
d=v+u
```

A función "obtenReg"

A implementación desta función realízase seguindo regras de selección dun rexistro para cada variable.





Pazo

XERACIÓN DE CÓDIGO ÓPTIMO PARA AS EXPRESIÓNS

mércores, 8 de maio de 2024 16:04

- Asignación de rexistros cando hai un número fixo deles
- Baseada nos **números de Ershov**
  - Indica cántos rexistros se necesitan para avaliar un nodo da árbore de expresión sen almacenar valores intermedios.

(a-b)+e*(c+d)

```
t1=a-b
t2=c+d
t3=e*t2
t4=t1+t3
```

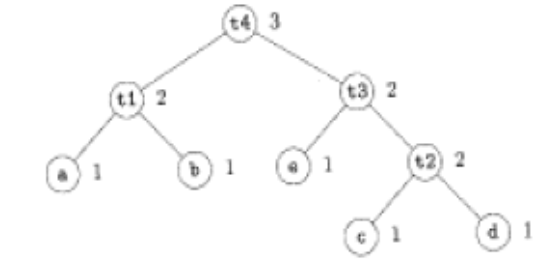


Figura 8.23: Un árbol etiquetado con números de Ershov

Xeración de Código a partir de Árbores de Expresión Etiquetadas

- Para xerar código sen almacenamento de valores temporais, empregando o número de rexistros indicado pola etiqueta da raíz da árbore.
- Segue o algoritmo 8.24.

p.e.:

```
t1=a-b
t2=c+d
t3=e*t2
t4=t1+t3
```

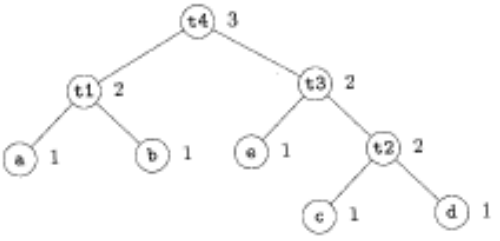


Figura 8.23: Un árbol etiquetado con números de Ershov

```
LD R3, d
LD R2, c
ADD R3, R2, R3
LD R2, e
MUL R3, R2, R3
LD R2, b
LD R1, a
SUB R2, R1, R2
ADD R3, R2, R3
```

Avaliación de expresións cun suministro insuficiente de rexistros

- É necesarios introducir instrucións de almacenamento.
- Segue o algoritmo 8.26.

p.e.:

```
t1=a-b
t2=c+d
t3=e*t2
t4=t1+t3
```

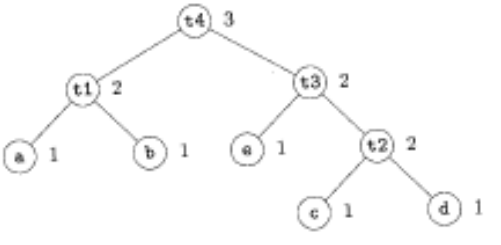


Figura 8.23: Un árbol etiquetado con números de Ershov

```
LD R2, d
LD R1, c
ADD R2, R1, R2
LD R1, e
MUL R2, R1, R2
ST t3, R2
LD R2, b
LD R1, a
SUB R2, R1, R2
LD R1, t3
ADD R3, R2, R1
```

Pazo